

# 1 Perfect is the Enemy of Good: Best-Effort 2 Program Synthesis

3 Hila Peleg 

4 University of California, San Diego, USA  
5 hpeleg@eng.ucsd.edu

6 Nadia Polikarpova 

7 University of California, San Diego, USA  
8 npolikarpova@eng.ucsd.edu

## 9 — Abstract —

---

10 Program synthesis promises to help software developers with everyday tasks by generating code  
11 snippets automatically from input-output examples and other high-level specifications. The conven-  
12 tional wisdom is that a synthesizer must always satisfy the specification exactly. We conjecture that  
13 this all-or-nothing paradigm stands in the way of adopting program synthesis as a developer tool: in  
14 practice, the user-written specification often contains errors or is simply too hard for the synthesizer  
15 to solve within a reasonable time; in these cases, the user is left with a single over-fitted result or,  
16 more often than not, no result at all. In this paper we propose a new program synthesis paradigm  
17 we call *best-effort program synthesis*, where the synthesizer returns a ranked list of partially-valid  
18 results, *i.e.* programs that satisfy some part of the specification.

19 To support this paradigm, we develop *best-effort enumeration*, a new synthesis algorithm that  
20 extends a popular program enumeration technique with the ability to accumulate and return multiple  
21 partially-valid results with minimal overhead. We implement this algorithm in a tool called BESTER,  
22 and evaluate it on 79 synthesis benchmarks from the literature. Contrary to the conventional wisdom,  
23 our evaluation shows that BESTER returns useful results even when the specification is flawed or too  
24 hard: *i)* for all benchmarks with an error in the specification, the top three BESTER results contain  
25 the correct solution, and *ii)* for most hard benchmarks, the top three results contain non-trivial  
26 *fragments* of the correct solution. We also performed an exploratory user study, which confirms our  
27 intuition that partially-valid results are useful: the study shows that programmers use the output of  
28 the synthesizer for comprehension and often incorporate it into their solutions.

29 **2012 ACM Subject Classification** Theory of computation → Program specifications; Software and  
30 its engineering → Automatic programming

31 **Keywords and phrases** Program Synthesis, Programming by Example

32 **Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2020.2

33 **Funding** This work has been supported by the National Science Foundation under Grant 1911149.

## 34 **1** Introduction

35 Program synthesis has emerged as a promising technology for automating low-level program-  
36 ming tasks [24, 50, 54, 3]. For software developers, program synthesis can be an attractive  
37 alternative to online help forums when it comes to “opportunistic programming” [11], or  
38 hunting for code that will perform a small subtask needed in a larger development task.  
39 Using a *Programming by Example* (PBE) synthesizer [36, 21, 20, 19, 46, 25, 56], developers  
40 can specify the desired behavior with a set of input-output examples (or unit tests), and the  
41 synthesizer would generate a code snippet that satisfies each of the examples.

42 Although PBE techniques have made great strides in recent years and have been used  
43 successfully in end-user tools [23, 31, 29], they have not seen wide adoption in mainstream  
44 software development. We conjecture that one important reason is that existing synthesizers  
45 follow an “all-or-nothing” paradigm: they either return a program that is correct on *all*



© Hila Peleg and Nadia Polikarpova;

licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 2; pp. 2:1–2:30

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

46 *examples*, or fail. In practice, however, humans make mistakes, so examples might contains  
 47 errors. Even if all the examples are correct, the program might just be too complex for the  
 48 synthesizer to generate: no matter how much we improve the synthesizer, there will always  
 49 be problems it fails to solve within the amount of time that the user is willing to wait. In  
 50 these cases, all-or-nothing synthesis is utterly useless to the programmer: it either returns a  
 51 single over-fitted result (that satisfies the erroneous specification) or, more often then not,  
 52 no result at all. Iterative synthesizers [32, 39, 7] offer a partial remedy by allowing the user  
 53 to refine a problematic specification, but they still waste user’s time in the unsuccessful  
 54 iterations.

55 We believe that turning PBE synthesizers into useful mainstream programming tools  
 56 requires addressing two core *challenges*:

- 57 1) **Erroneous specifications:** How can we make the synthesizer robust to small errors in  
 58 the specification?
- 59 2) **Hard problems:** How can we make the synthesizer helpful even if it cannot solve a  
 60 problem completely?

### 61 Switching paradigms

62 To address the two core challenges, we need to abandon the all-or-nothing view of synthesis  
 63 and instead take the approach of successful code completion tools: an imperfect result is  
 64 better than no result, as long as it is indicated as such. To this end, we propose a new PBE  
 65 paradigm we dub *best-effort program synthesis*, in which the user provides examples, and the  
 66 synthesizer returns a shortlist of partially-valid results, *i.e.* programs that satisfy at least  
 67 some of the examples. Previous work has shown that *a)* partially-valid programs often share  
 68 non-trivial fragments with the correct solution [46], and *b)* users prefer editing incorrect  
 69 code to writing code from scratch [13]. Hence it is reasonable to assume that partially-valid  
 70 results help the user move forward both when the specification contains errors (by generating  
 71 a solution for the error-free subset of the examples) and when the problem is too hard (by  
 72 generating a spacial-case program that can be used as a building block in the final solution).

### 73 Efficient best-effort synthesis

74 A naive way to implement best-effort synthesis would be to use an existing synthesizer as a  
 75 black box and re-run it again and again with different subsets of the specification, displaying  
 76 any generated programs to the user. This is highly inefficient, however, especially when the  
 77 original synthesis problem takes too long to solve: in this case, some specification subsets may  
 78 still take too long. Ideally, we would like to deliver partially-valid results without requiring  
 79 the synthesizer to do more work.

80 Our core *technical insight* is that a popular program search algorithm—bottom-up  
 81 enumeration with observational equivalence reduction [55, 2]—can be extended to accumulate  
 82 partially-valid results during search with minimal overhead. The extension is possible because  
 83 this search algorithm is *monotonic* in the set of examples: the set of programs explored with  
 84 the full specification includes all programs that would be explored with a partial specification.  
 85 We formalize this monotonicity property and our extended *best-effort enumeration* algorithm  
 86 in Section 3.

## 87 Ranking partially-valid results

88 In general, there can be too many partially-valid results to display them all to the user, so a  
89 best-effort synthesizer needs a way to automatically select a manageable number of results  
90 (3–5) that are most likely to be useful to the programmer. It is common in program synthesis  
91 to introduce a *ranking function* for the generated programs and present top  $N \geq 1$  ranked  
92 results to the user [23, 28, 43]. For the best-effort setting, we design a ranking function that  
93 incorporates both *syntactic* and *semantic* features of programs, such as simplicity and the  
94 number of examples satisfied. The details of the ranking are described in Section 4.

## 95 Evaluating best-effort solutions

96 We implement our approach in a tool called BESTER (Best-Effort Synthesis TERminal), which  
97 gives users access to a best-effort synthesizer from a Read-Evaluate-Print-Loop (REPL). We  
98 evaluate BESTER on 79 benchmarks we collected from the 2017 SYGUS competition [4] and  
99 the EUPHONY benchmark suite [33]. Our evaluation shows that *i*) BESTER can overcome errors  
100 in the specification and still return the correct solution in the top three results, *ii*) when a  
101 synthesis problem is hard and times out, BESTER still returns useful fragments of the solution,  
102 and *iii*) BESTER’s ability to solve correct specifications is not impacted (Section 5). Moreover,  
103 BESTER compares favorably to the naive approach of using a state-of-the-art synthesizer<sup>1</sup> as a  
104 black box and eliminating examples from the specification one by one.

105 We also performed a small exploratory user study of BESTER, in which programmers used  
106 BESTER to solve tasks in an unfamiliar programming language; the tasks were too hard for  
107 the synthesizer to solve completely within 40 seconds (Section 6). Our study shows that  
108 programmers make use of synthesis results for comprehension, both of the task and of the  
109 language, and that programmers often incorporate synthesis results into their solutions either  
110 by copy-pasting or by editing a partially-valid solution until it fully satisfies the examples.

## 111 Main contributions

112 To summarize, this paper makes the following contributions:

- 113 1. *Best-effort program synthesis*: a new user interaction paradigm for PBE that is likely to  
114 yield helpful results even when the problem is ill-specified or too hard to solve completely.
- 115 2. *Best-effort enumeration*: an algorithm for efficiently collecting partially-valid solutions  
116 during enumerative synthesis.
- 117 3. A ranking function for partially-valid solutions that incorporates both syntactic and  
118 semantic properties of programs, and performs well in our experiments.
- 119 4. BESTER: a prototype implementation of best-effort synthesis, shown both empirically and  
120 in an exploratory user study to be robust to specification errors and to produce useful  
121 program fragments on hard problems.

## 122 2 Overview

123 In this section, we consider a scenario that requires best-effort synthesis.

---

<sup>1</sup> We used CVC4 [44], the winner of the 2017–2019 SYGUS competitions in the PBE-Strings category.

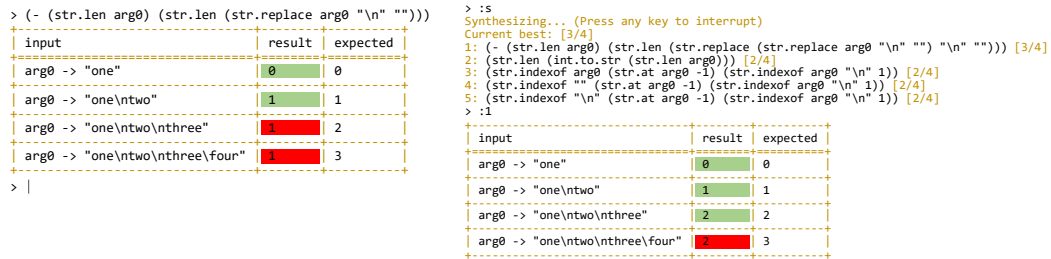


Figure 1 The BESTER REPL interface.

## 2.1 A motivating example

Our example is derived from one of the benchmarks in the PBE-Strings track of the SYGUS (Syntax-Guided Synthesis) competition [5, 4]. In this competition, synthesizers are expected to generate programs in a simple language of S-expressions with built-in operations on integers (such as + or -) and strings (such as `str.len` and `str.replace`). A benchmark in the PBE-Strings track is given by a set of input-output examples and a grammar that defines the space of candidate programs (*i.e.* the relevant subset of the SYGUS language). These benchmarks mimic small tasks performed by programmers, and some are directly derived from StackOverflow questions.

In this scenario, a programmer is attempting to solve a task that asks them to count the number of line breaks in a string. They are using a development environment enriched with a synthesizer: they have the option to invoke the synthesizer at any point during development and incorporate (fragments of) its output into their own code.

The programmer starts by providing a set of test cases (examples):

```
e0 = "one" → 0
e1 = "one\ntwo" → 1
e2 = "one\ntwo\nthree" → 2
e3 = "one\ntwo\nthree\nfour" → 3
```

We notice, though the user does not, that  $e_3$  contains a typo in the string and would, given the expected program, only return 2 rather than 3.

The user then attempts to write a program to satisfy their test cases by computing the difference in length between the input string, `arg0`, and `arg0` with newlines removed:

```
(- (str.len arg0) (str.len (str.replace arg0 "\n" "")))
```

The user executes their tests, and only  $e_0$  and  $e_1$  pass, as shown in Figure 1a. They might not immediately realize that the reason for this behavior is the unexpected semantics of `str.replace` in the SYGUS language, which only replaces the first instance of the substring rather than all instances. Because  $e_2$  fails as well as  $e_3$ , the typo in  $e_3$  goes unnoticed.

At this point, the user decides to delegate solving the task to the synthesizer. Running the state-of-the-art synthesizer CVC4 [44] on this synthesis query yields the result:

```
(ite (str.contains (str.replace arg0 "\n" "") "\n")
      (ite (str.suffixof (str.at arg0 (str.len "\n")) arg0)
            (str.len "\n") (str.indexof arg0 "\n" 1))
      (ite (str.prefixof arg0 (str.replace arg0 "\n" arg0)) 0 1))
```

151 This program satisfies all the test cases provided to the synthesizer, but it is so complex that  
 152 the user will most likely discard it without reading and be none the wiser about the typo in  
 153 the tests or their misconception about the semantics of `str.replace`.

154 Running our tool BESTER, on the other hand, produces a ranked list of synthesis results,  
 155 as shown in Figure 1b. The first result in this list is:

```
156 (- (str.len arg0) (str.len (str.replace (str.replace arg0 "\n" "") "\n" "")))
```

157 which is relatively simple and in fact similar to the user’s initial solution (except that it  
 158 calls `str.replace` on the input string twice). Contrasting the outputs of the initial program  
 159 and this result helps the user realize their misconception about `str.replace`, while the tool’s  
 160 failure to solve  $e_3$  is likely to call their attention to the typo.

### 161 Best-effort synthesis for hard specifications

162 Consider a slightly different specification our programmer could have provided, where  
 163 examples  $e_0, e_1, e_2$  are as before, but example  $e_3$  is replaced with

```
164 e'_3 = "one\two\three\four\nfive\nsix\nseven\neight" → 7
```

165 The programmer asks the (traditional) synthesizer for help, but after 30 seconds of waiting,  
 166 their patience is exhausted, and they interrupt the synthesizer before it can produce any  
 167 results. The reason this problem is taking so long to solve is that the SyGuS language  
 168 contains no general solution that works for an arbitrary number of newlines, so the shortest  
 169 program that satisfies  $e'_3$  contains seven calls to `str.replace`; programs of this size present a  
 170 challenge for state-of-the-art synthesizers. Once again, the user just wasted their time and is  
 171 back to square one.

172 Although this particular example seems contrived, the general scenario where the user is  
 173 unaware of the limitations of the synthesis algorithm and gives it more than it can handle,  
 174 is very common. If the programmer is using BESTER, however, and interrupts it after 30  
 175 seconds, they would get exactly the same set of results as in the previous scenario, shown  
 176 in Figure 1b. This is because BESTER always searches for solutions to all subsets of input  
 177 examples simultaneously, and the solution for  $\{e_0, e_1, e_2\}$  is much smaller—and hence will be  
 178 discovered much earlier—than the solution for the full set of examples.

## 179 2.2 Background: Observational Equivalence Reduction

180 Before we explain how BESTER is able to generate such partially-valid results efficiently, we  
 181 must introduce the baseline synthesis technique we build upon: bottom-up enumeration with  
 182 *observational equivalence reduction* [55, 2], or OE-reduction for short. Program synthesizers  
 183 work by searching a space of candidate programs until they encounter one that satisfies the  
 184 specification. The central challenge of program synthesis is the astronomically large size of  
 185 the search space, so different synthesis techniques find different ways to reduce the space, *i.e.*  
 186 exclude large chunks of the space from consideration.

187 For illustration purposes, in this section we will consider the program space defined by an  
 188 artificially small grammar, shown in Figure 2a. This grammar allows using only two integer  
 189 literals (0 and 3), one string literal (" "), a single variable (`input`), and three operations: `+`,  
 190 `str.indexof`, and `str.substr`.

191 **Bottom-up enumeration**

192 Bottom-up enumeration is a synthesis technique that maintains a *bank* of enumerated  
 193 programs and constructs new programs by applying production rules to programs from the  
 194 bank. Recall the grammar in Figure 2a. We begin enumeration with an empty bank, so in  
 195 the first iteration we are limited to production rules that require no subexpressions—literals  
 196 and variables; this yields the programs 0, 3, " ", and `input`, which are added to the bank.  
 197 In the following iterations, production rules that require subexpressions are applied to the  
 198 programs in the bank: for example, the rule  $Int \rightarrow (+ Int Int)$  is applied to all pairs of  
 199 *Int* expressions, creating new programs  $(+ 0 0)$ ,  $(+ 0 3)$ ,  $(+ 3 0)$ , and  $(+ 3 3)$ , as seen in  
 200 Figure 2b.

201 The enumeration is generally performed in the order of height: we first construct all  
 202 programs of height 0, then height 1 and so on; each iteration constructs all programs of  
 203 height  $n + 1$  using the programs of heights up to  $n$  stored in the bank. As a consequence,  
 204 discarding even a few programs from the bank can drastically reduce the number of programs  
 205 to be enumerated in future iterations.

206 **Equivalence reduction**

207 A natural candidate for discarding from the bank is a redundant program, *i.e.*, a program  
 208 that is functionally equivalent to another program in the bank. In our example, the program  
 209  $(+ 0 3)$  is functionally equivalent to the program 3, and hence can be safely discarded.  
 210 State-of-the-art bottom-up synthesizers [55, 2, 6] use a more aggressive notion of program  
 211 equivalence called *observational equivalence*, which is also easier to check: two programs are  
 212 considered equivalent if they evaluate to the same output for every input in the user-provided  
 213 set of examples.

214 ► **Example 1.** Let us assume two pairs of input-output examples

215  $e_0 = \text{"The Demolished Man"} \rightarrow \text{"Demolished"}$   
 $e_1 = \text{"The Stars My Destination"} \rightarrow \text{"Stars"}$

216 We follow the enumeration of programs with OE-reduction, summarized in Figure 2b.

First, we create an *input vector*, which in this case contains two inputs:

$\langle \text{"The Demolished Man"}, \text{"The Stars My Destination"} \rangle$

217 The algorithm evaluates each constructed program point-wise on the input vector, producing  
 218 an *output vector*. Two programs are deemed observationally equivalent if their output vectors  
 219 are equal.

220 **Height 0:** First we enumerate programs of height 0 (programs 1–4 in Figure 2b). The pro-  
 221 gram 0 is a literal and evaluates to 0 on every input, resulting in the output vector  $\langle 0, 0 \rangle$ . Like-  
 222 wise the programs 3 and " " result in  $\langle 3, 3 \rangle$  and  $\langle \text{" "}, \text{" "} \rangle$  respectively. The program `input` (the  
 223 input variable) yields the output vector  $\langle \text{"The Demolished Man"}, \text{"The Stars My Destination"} \rangle$ .  
 224 Since all four output vectors are different, all four programs are added to the bank.

225 **Height 1:** Next, we enumerate programs of height  $n + 1$  by applying production rules  
 226 in the grammar to programs from the bank at heights up to  $n$  (in this case, up to 0). The  
 227 production rule for `str.indexof` requires two arguments of type string, and will be applied to  
 228 all combinations of string programs of height 0. This will produce, among others, the program  
 229  $(\text{str.indexof } \text{" " } \text{" "})$  with the output vector  $\langle 0, 0 \rangle$ . Notice that the bank already contains  
 230 a program with this vector: the program 0. The algorithm therefore discards  $(\text{str.indexof}$

```

Start → String
Int   → 0 | 3
      | (+ Int Int)
      | (str.indexof String String)
String → " " | input
       | (str.substr String Int Int)

```

(a) A small grammar in the SYGUS format. Notice that the language is limited to the literal constants that appear here.

#	program	output on $e_0$	output on $e_1$	equivalent to
1	0	0	0	
2	3	3	3	
3	" "	" "	" "	
4	input	"The Demolished Man"	"The Stars My Destination"	
5	<del>(+ 0 0)</del>	0	0	#1
6	<del>(+ 0 3)</del>	3	3	#2
7	<del>(+ 3 0)</del>	3	3	#2
8	(+ 3 3)	6	6	
9	<del>(str.indexof " " " ")</del>	0	0	#1
10	<del>(str.indexof " " input)</del>	-1	-1	
11	<del>(str.indexof input input)</del>	0	0	#1
12	<del>(str.indexof input " ")</del>	3	3	#2

(b) An enumeration of the grammar by height.

■ **Figure 2** The enumeration in Example 1. Programs are generated from the grammar by height, first productions requiring only a terminal, and next productions requiring a subtree, taken from previously seen programs.

231 " " " ") and does not add it to the bank. In general, the algorithm maintains an *invariant*  
 232 that the bank contains at most one representative of any observational equivalence class.

233 The same production rule also generates the program (str.indexof input " "). This pro-  
 234 gram seems helpful for solving the given examples; however, its output vector is  $\langle 3, 3 \rangle$ , whose  
 235 equivalence class already has a representative, the program 3, so the program (str.indexof  
 236 input " ") will be discarded. Unlike in the case of (str.indexof " " " "), this seems an  
 237 imprudent decision. However, it is in fact sound to do so *for these inputs*: so long as we do  
 238 not care about differently structured inputs, (str.indexof input " ") and 3 are completely  
 239 interchangeable. If the user introduces another example with a new input such as "Virtual  
 240 Unrealities", the new extended output vectors will be  $\langle 3, 3, 3 \rangle$  and  $\langle 3, 3, 7 \rangle$ , and the two  
 241 programs will no longer be equivalent.

## 242 2.3 Our approach

243 Next we describe how BESTER modifies the baseline OE-reduction enumeration technique from  
 244 the previous subsection to maintain a ranked list of partially-valid programs. If the search  
 245 happens to encounter a program that fully satisfies the specification, it stops; otherwise, if  
 246 the search is interrupted before a solution was found, BESTER simply returns the current list  
 247 of partially-valid results to the user. We refer to this modification of OE-reduction search as

248 *best-effort enumeration*; Section 3 details the search algorithm and its correctness.

### 249 Searching for all example subsets

250 Recall the task from Section 2.1, where the user is trying to count line breaks in a string,  
251 but has an error in the example  $e_3$ . We would like to show the programmer the following  
252 partially-valid yet useful program  $p^*$ , which satisfies examples  $\{e_0, e_1, e_2\}$ :

```
253 (- (str.len arg0) (str.len (str.replace (str.replace arg0 "\n" "") "\n" "")))
```

254 Since we do not know a-priori which subset of examples would yield a useful result, we would  
255 like the synthesizer to simultaneously search for programs satisfying *all* non-empty subsets  
256 of  $\{e_0, e_1, e_2, e_3\}$  (thus, including  $\{e_0, e_1, e_2\}$ ).

257 Note that many synthesis techniques are not amenable to such simultaneous search: for  
258 example, in constraint-based synthesis [52, 26], a run of the synthesizer with the full set of  
259 examples would never construct  $p^*$ , because it does not satisfy  $e_3$ . We observe that unlike  
260 most synthesis techniques, the OE-reduction algorithm has the ability to maintain solutions  
261 for all example subsets with little to no overhead, thanks to a curious monotonicity property:  
262 *adding a new example never excludes programs from the enumeration.*

263 Let us illustrate this property on our running example. Consider a *hypothetical* run of an  
264 OE-synthesizer on the examples  $\{e_0, e_1, e_2\}$ , and assume that in this run  $p^*$  is added to the  
265 bank. We conclude that  $p^*$  is the first program the synthesizer constructed that produces  
266 the output vector  $\langle 0, 1, 2 \rangle$ , and hence has been chosen as the representative of the  $\langle 0, 1, 2 \rangle$   
267 equivalence class. Now consider the *actual* run of the synthesizer, on the full set of examples  
268  $\{e_0, e_1, e_2, e_3\}$ ; we argue that in this run  $p^*$  must be chosen as the representative of the  
269  $\langle 0, 1, 2, 2 \rangle$  equivalence class and cannot be discarded by OE reduction. To see why, assume a  
270 different program  $p'$  is chosen as the representative; then  $p'$  would have been enumerated  
271 before  $p^*$  and would also return  $\langle 0, 1, 2 \rangle$  on the first three examples; but this contradicts our  
272 assumption that  $p^*$  is the representative for  $\langle 0, 1, 2 \rangle$ .

273 In other words, since each additional example *refines* the partition of the program space,  
274 the bank in the actual run must be a superset of the bank in the hypothetical run. Moreover,  
275 the output vector of each program in the bank is already computed as part of performing  
276 OE-reduction, and compared to the expected output vector; hence, performing a slightly  
277 more complex check for the purpose of identifying partially-valid results incurs only minimal  
278 overhead.

### 279 Ranking best-effort candidates

280 A best-effort enumeration as described above might accumulate multiple results satisfying  
281 each subset of the examples. However, we cannot simply show them to the user in the  
282 order in which they are discovered: trivial programs such as a literal or variable satisfying  
283 one or two of the examples would be discovered immediately, but would often be a poor  
284 candidate. For instance, in the example from Section 2.1, the program 0 satisfies  $\{e_0\}$ , the  
285 program `(str.indexof arg0 "\n")` satisfies  $\{e_4\}$  (the erroneous example), and the program  
286 `(ite (str.contains arg0 "\n") 1 0)` satisfies  $\{e_0, e_1\}$ . All of these will be discovered fairly  
287 early on in the enumeration.

288 Instead, the partially-valid programs in the bank need to be *ranked* so that a manageable  
289 number (no more than 5) of promising programs can be returned to the user. We have  
290 developed a simple ranking function for BESTER that takes into account both syntactic and  
291 semantic properties of programs, and performs well empirically. Section 4 details our ranking



292 function and discusses other possible rankings. Intuitively, our ranking rewards programs  
 293 that satisfy more examples, programs that use all of their inputs (the so called *relevancy*  
 294 *requirement* inspired by other synthesis techniques [20, 27]), smaller programs, and programs  
 295 where the incorrect outputs are close to the expected outputs. Among the programs listed  
 296 above, `(str.indexof arg0 "\n")` and `0` both satisfy one example, but the former is preferred  
 297 by our ranking because it uses its input.

### 298 **3 Best-Effort Enumeration With Observational Equivalence**

299 In this section, we detail the way an enumerative search with observational equivalence can  
 300 be used to find and rank best-effort results to a synthesis query.

301 Let us consider the challenge in finding a best-effort solution. Since the set of user-  
 302 provided examples  $\mathcal{E}$  might be unsatisfiable, we wish to return a program that satisfies some  
 303  $\mathcal{E}^* \subseteq \mathcal{E}$ . However, we do not know in advance whether  $\mathcal{E}$  is satisfiable, and if it is not, *which*  
 304  $\mathcal{E}^*$  we are searching for a solution to.

305 We can address this challenge with minimal effort thanks to several properties of equiva-  
 306 lence classes.

#### 307 **Refined equivalence classes**

308 Enumerative synthesis with observational equivalence adds only one representative from each  
 309 equivalence class to its bank of programs based on an equivalence relation  $\equiv_I$  defined as  
 310 follows:

$$311 \quad p_1 \equiv_I p_2 \iff \forall \iota \in I. \llbracket p_1 \rrbracket(\iota) = \llbracket p_2 \rrbracket(\iota)$$

312 where the equality of execution results considers outputs, exceptions, and side effects. In  
 313 a PBE synthesis query, the inputs in  $I$  are derived from the example set  $\mathcal{E}$  such that  
 314  $I = \{\iota \mid (\iota, \omega) \in \mathcal{E}\}$ .

315 If the enumeration that has already added to the reduced program bank the program  
 316  $p$  encounters a program  $p'$  such that  $p \equiv_I p'$ , a decision is made which one will be the  
 317 *representative of the equivalence class*  $[p]$  that both  $p$  and  $p'$  inhabit. The representative is  
 318 then kept in the program bank and the other program is discarded. In most synthesizers that  
 319 perform the enumeration in layers (i.e., first programs of height 0, then of height 1, etc.), the  
 320 first program encountered from each equivalence class is selected as its representative, as was  
 321 shown in Figure 2b.

322 Now consider  $\mathcal{E}' \subset \mathcal{E}$ , a non-empty subset of examples, and its input set  $I'$ . It is easy to  
 323 see that  $\equiv_I$  is a *refinement* of  $\equiv_{I'}$ , since it is the intersection of  $\equiv_{I'}$  and  $\equiv_{I \setminus I'}$ . This means  
 324 that  $\equiv_I$  refines the partition into equivalence classes made by  $\equiv_{I'}$ , or that for a program  $p$   
 325 in the candidate program space,  $[p]_{\equiv_I} \subseteq [p]_{\equiv_{I'}}$ .

326 We notice that if selection of the representative is deterministic, then if  $p$  was the  
 327 representative of  $[p]_{\equiv_{I'}}$ , the less refined (and possibly larger) equivalence class, then  $p$   
 328 will also be the representative of  $[p]_{\equiv_I}$ : representative selection has determined  $p$  to be  
 329 the representative against each of the programs in  $[p]_{\equiv_I}$  when it was decided to be the  
 330 representative of  $[p]_{\equiv_{I'}}$ .

331 This means that if  $p$  was included in the bank of programs in a less refined enumeration  
 332 with OE-reduction,  $p$  will be in the program bank of a more refined enumeration, one with  
 333 more examples.

■ **Algorithm 1** A best-effort enumeration

---

```

Input:  $\mathcal{E}$  a user-provided example specification,  $\mathcal{G}$  a grammar,  $f$  a fitness function,
           $maxResults$  the maximum number of results to return to the user
Result: Top  $maxResults$  synthesized programs
1   $programBank \leftarrow \emptyset$ 
2   $resultCandidates \leftarrow PriorityQueue()$ 
3  while timeout has not passed do
4    foreach  $prodRule \in \mathcal{G}$  do
5       $k \leftarrow arity(prodRule)$ 
6      foreach  $(arg_1, \dots, arg_k) \in programBank^k$  do
7        if  $(arg_1, \dots, arg_k)$  is suitable for prodRule then
8           $newProg \leftarrow prodRule(arg_1, \dots, arg_k)$ 
9          if  $\forall p \in programBank. p \not\equiv_I newProg$  then
10             /* Found the representative of a new equivalence class,
11             add to the bank */
12              $programBank \leftarrow programBank \cup \{newProg\}$ 
13              $exec \leftarrow \{(\iota, \llbracket newProg \rrbracket(\iota)) \mid \iota \in I\}$ 
14             if  $exec \cap \mathcal{E} \neq \emptyset$  then /* newProg partially satisfies  $\mathcal{E}$  */
15              $resultCandidates.insertWithPriority(newProg, f(newProg, \mathcal{E}))$ 
16             end
17             if  $exec = \mathcal{E}$  then /* newProg fully satisfies  $\mathcal{E}$  */
18              $break$  all loops
19           end
20         end
21       end
22     end
23     /* Either timeout has passed and or a fully satisfying program was
24     found. We now return a list of options by rank. */
25      $results \leftarrow List()$ 
26     for  $i = 1$  to  $\min(maxResults, resultCandidates.size())$  do
27        $results.append(resultCandidates.getFront())$ 
28     end
29     return  $results$ 

```

---

334 Notice that, despite the use of an inputs *vector* in Section 2.2 (and in practical imple-  
335 mentations), the operations are unordered. This means that it does not matter which of the  
336 examples are missing from  $\mathcal{E}^-$  for the property to hold.

### 337 3.1 Finding best-effort solutions

338 Fortunately, since performing observational equivalence with  $\mathcal{E}$  is a refinement of any strict,  
339 nonempty subset of  $\mathcal{E}$ , we can essentially test all nonempty subsets of  $\mathcal{E}$  simultaneously.  
340 Representative selection ensures we will see all programs we would see enumerating a subset  
341 of the examples, so we can simply collect programs that satisfy any of the examples, instead  
342 of ones that satisfy *all* of them.

343 Lines 4 – 8 of Algorithm 1 are a simple bottom-up enumeration of the space, applying

344 each of the production rules to each of the programs previously added to the program bank,  
 345 generating additional programs. Lines 9 – 10 are the implementation of the OE-reduction,  
 346 adding to the program bank only programs that are the first of their equivalence class to be  
 347 encountered. Line 15 is the stopping condition for any PBE synthesizer: whether executing  
 348 each input leads to its expected output. It is simply lines 12 – 14 that “piggyback” on  
 349 the enumeration with observational equivalence, collecting programs that satisfy any of the  
 350 examples and create the best-effort search.

351 This means that when enumerating the example in Section 2.1, the program

```
352 (- (str.len arg0) (str.len (str.replace (str.replace arg0 "\n" "") "\n" "")))
```

353 is produced by the algorithm on line 8. In a regular observational equivalence reduction, the  
 354 program will be added to the reduced program bank on line 10 for use in enumerating larger  
 355 programs, and the next step would be to perform the check on line 15, testing whether it  
 356 fully satisfies the specification. Since it satisfies 3 of the 4 examples, a simple enumeration  
 357 would not return it and enumeration would continue searching for a single fully-satisfying  
 358 program to show the user.

359 In a best-effort enumeration, the condition on line 12 admits programs that satisfy any  
 360 nonempty subset of  $\mathcal{E}$ . The program is added to the list of best-effort results, of which the  
 361 best results will be returned to the user.

362 The correctness proposition of observational equivalence [2] guarantees that if a program  
 363 that satisfies  $\mathcal{E}$  exists in the space, we will encounter exactly one such program, as other  
 364 programs satisfying  $\mathcal{E}$  are in its equivalence class and are not part of the reduced program space.  
 365 However, if we consider any strict subset, this guarantee no longer holds: when partitioning  
 366 the space of programs possible in the grammar based on observational equivalence for  $\mathcal{E}$ , any  
 367  $\mathcal{E}' \subset \mathcal{E}$  is now represented by a number of equivalence classes in the program space instead  
 368 of just one. In other words, more than one program satisfying  $\mathcal{E}'$  may be encountered in the  
 369 course of the enumeration.

370 This means there are two dimensions in which our goal is no longer unique: along an  
 371 enumeration, we are looking for a program that satisfies one of exponentially many  $\mathcal{E}' \subseteq \mathcal{E}$ ,  
 372 and there can be many such programs for each  $\mathcal{E}'$ . However, since the results of a best-effort  
 373 enumeration are intended for consumption by a user, we must limit ourselves to a small  
 374 number of returned results. This means that in the course of an enumeration based on  $\mathcal{E}$   
 375 programs that satisfy any nonempty subset of  $\mathcal{E}$  are collected, and the *best* few are returned  
 376 to the user. This is determined by a fitness function used to rank the programs in line 13 of  
 377 Algorithm 1.

378 We will introduce our fitness function in the next section.

## 379 **4 Fitness Function**

380 As we have shown in Sections 2 and 3, more than one program can satisfy the same number  
 381 of specifications. In this section, we discuss the considerations in constructing the fitness  
 382 function used in our implementation of BESTER, and suggest additional parameters that  
 383 could be added for other synthesizers.

384 The composition of the function is:

$$385 f(p, \mathcal{E}) = 3 \cdot \text{satisfied}(p, \mathcal{E}) + 2 \cdot \text{relevancy}(p) + \text{distance}(p, \mathcal{E}) + \text{size}(p)$$

386 We now break down each of these elements.

387 **Examples satisfied**

388 Since a program satisfying one example and a program satisfying all examples but one are  
389 not equally good, we use the portion of examples satisfied in our ranking of the program.

$$390 \quad \text{satisfied}(p, \mathcal{E}) = \frac{|\{(\iota, \omega) \in \mathcal{E} \mid \llbracket p \rrbracket(\iota) = \omega\}|}{|\mathcal{E}|}$$

391 This portion of the fitness function is the most strongly weighted, as we still give the  
392 most importance to the best effort, *i.e.* solving the largest portion of the specification.

393 **Relevancy**

394 Given two programs that solve the same number of examples, we prefer one that uses more  
395 of its input. For example, let us assume a grammar with two input variables, `arg0` and `arg1`,  
396 and three programs that satisfy 2 of 3 examples in  $\mathcal{E}$ :

```
397   p1 = true
398   p2 = (str.contains arg0 " ")
399   p3 = (str.prefixof arg1 arg0)
```

398 Intuitively, we are certain we want  $f(p_1)$  to be the lowest of the three, but in all likelihood,  
399 we also want to reward  $p_3$  for using all available input from the user. This is a tactic employed  
400 by other synthesis tools such as [20, 27].

401 We define for all variables  $\mathcal{V}$  available in the grammar:

$$402 \quad \text{relevancy}(p) = \frac{|\{var \in \mathcal{V} \mid var \in p\}|}{|\mathcal{V}|}$$

403 **Distance from output**

404 While we strongly reward a program for each satisfied example, we also wish to reward  
405 programs that do “better” with regard to the remaining examples.

406 Currently we include this element only for synthesis tasks that search for a string program.  
407 For strings, being closer to the expected output can be seen as returning a subset or superset  
408 of it, or constructing a close string. This is easily rewarded by using Levenshtein Distance [34]  
409 to measure the distance of the *unsatisfied* example results from the intended output. While  
410 this component may not be suitable for numeric types, for other structured types such as  
411 lists or trees, other such structured distance metrics can be employed in place of *LD*.

412 We denote  $\mathcal{E}^- = \{(\iota, \omega) \in \mathcal{E} \mid \llbracket p \rrbracket(\iota) \neq \omega\}$  to be the unsatisfied examples, and define:

$$413 \quad \text{distance}(p, \mathcal{E}) = \begin{cases} \text{avg}_{(\iota, \omega) \in \mathcal{E}^-} \left( \left\{ 1 - \frac{LD(\llbracket p \rrbracket(\iota), \omega)}{\max(|\omega|, |\llbracket p \rrbracket(\iota)|)} \right\} \right) & p \text{ is a string program and } |\mathcal{E}^-| > 0 \\ 0 & \text{o.w.} \end{cases}$$

414 While we include this in the fitness function, we do not weight it as high as some of the  
415 other components as we do still want to allow other logic that may help the user toward the  
416 correct answer, e.g., constructing a complement of the result in order to remove it, to rank  
417 well and be displayed.

## 418 Program size

419 Finally, we incorporate the size of the program into the function. In a regular enumerative  
 420 synthesizer, ranking by size is implicit, as programs of a lower height will be reached first.  
 421 Since programs of a lower height are *simpler programs*, this tactic is employed in many  
 422 synthesizers. In best-effort synthesis we may encounter programs of very different sizes that  
 423 satisfy the same examples before we reach the timeout. We therefore add the height of the  
 424 program into the ranking to prefer shorter ASTs.

425 Additionally, we would like to distinguish between programs of the same height. To do  
 426 this, we use  $terms(p)$ , the number of nodes in the AST of  $p$ . For example,  $p_1 = (\text{str.at arg0}$   
 427  $(+ 1 1))$  and  $p_2 = (\text{str.++ (str.++ " " " ") (str.substr arg0 1 1))$  are both programs of  
 428 height 2, but  $terms(p_1) = 5$  whereas  $terms(p_2) = 8$ .

429 Since programs are eventually displayed to a user, given two programs of the same height  
 430 that are indistinguishable by other parameters, we would like to show the user first the one  
 431 that is easier to read, or the overall-smaller one.

432 Together, we define:

$$433 \quad size(p) = \frac{1}{height(p) + 1} + \frac{1}{terms(p)}$$

## 434 Including other data

435 In a domain where not all specifications are created equal, some may be ranked as more  
 436 important than others. For instance, examples that detail an error scenario may be deemed  
 437 more or less important than examples that specify a simple output value. Likewise, if not all  
 438 specifications are examples [40], an importance ranking between different specification types  
 439 can be used to decide which are more likely to be dropped.

440 Finally, we address the fact that our fitness function is not learned. In theory, a model  
 441 could be trained to compute a fitness function according to desired program rankings, or to  
 442 provide features for a fitness function (e.g., [8, 33] compute the probability of a program,  
 443 which in their tool is used to speed up the search but could also be used for simple numerical  
 444 ranking). However, the pool of programs is small, and creating a dataset of ranked best-effort  
 445 programs large enough to train from, either manually or automatically, would be unreliable  
 446 at best. In addition, our fitness function, both in selected features and in their weights,  
 447 encodes in it what we consider to be the important aspects of a best-effort program, rather  
 448 than numbers overfitted to a small dataset.

## 449 **5 Empirical Evaluation**

450 In this section we detail the empirical evaluation performed to validate our approach. Our  
 451 experiments are based on the benchmarks of the SYGUS competition [4] and EUPHONY [33].

### 452 Implementation

453 We implemented an enumerating, observational equivalence synthesizer for the SYGUS  
 454 language in Scala, then augmented it for best-effort enumeration<sup>2</sup>. Best-effort solutions  
 455 are accumulated as the enumeration progresses, and the top 5 results are returned. The

<sup>2</sup> <https://github.com/peleghila/bester>

456 enumeration loop of our synthesizer has a 40s timeout, selected since it is a manageable  
457 length of task interruption for a human user [37].

## 458 **Benchmarks**

459 We used a set of 79 synthesis queries from the 2017 SYGUS competition and the EUPHONY  
460 benchmarks. These benchmarks contain a selection of data wrangling and string transforma-  
461 tion tasks: the SYGUS benchmarks are entirely string to string transformations but 19 of the  
462 EUPHONY benchmarks either have a non-string parameter or synthesize a numeric or boolean  
463 expression. Duplicate tasks between SYGUS and EUPHONY were removed from the original  
464 benchmark set, as well as benchmarks requiring recursion.

465 We initially divided them into two sets using a simple OE-based enumerating synthesizer  
466 (that does not collect best-effort results): 63 that can be solved within 40s, denoted “easy”,  
467 and 16 that cannot, denoted “hard”.

468 We then created a modified version of the benchmarks in the “easy” set by adding  
469 erroneous examples such as typos, off-by-one errors, etc. This was done manually and  
470 required great care in order to make sure that the additions are *i*) not consistent with  
471 the original target program, and *ii*) do not always create a new example set that is easily  
472 generalized. Of 37 modified benchmarks, two contain more than one erroneous example.

473 We note that while we introduced errors, it is near impossible to introduce *contradictions*,  
474 short of pairing the same input with two different outputs. Since most of the SYGUS and  
475 EUPHONY benchmarks include the conditional `ite` in their grammar, given enough time the  
476 inconsistency in the examples in many of the modified benchmarks can be overcome with  
477 case-splitting. The exception to this is a result that requires string constants not included in  
478 the grammar and that cannot be generated from the input.

479 For convenience, we use the simple OE synthesizer to make a distinction between the  
480 modified benchmarks:

- 481 1. “no-solution”: benchmarks in which the synthesizer does not find a program that satisfies  
482 all examples within the 40s timeout, and
- 483 2. “overfitted”: benchmarks in which the synthesizer is able to find a solution to the given  
484 examples (this solution will usually be long and overfitted via multiple case splits).

485 Since the origin of many of our benchmarks is the `PBE-Strings` track of the SYGUS  
486 competition, we take as state-of-the-art the synthesizer/solver CVC4 [44], winner of the  
487 `PBE-Strings` track of the competition since 2017. We use CVC4 1.7, the most recent version  
488 available.

## 489 **Experimental setup**

490 We generated gold-standard solutions for each of the original, unmodified 79 benchmarks. Our  
491 gold standard is more forgiving than the SYGUS competition, including both hand-written  
492 solutions for the task in the benchmark, as understood by the authors, and solutions from  
493 CVC4 that cover all examples, despite taking a different approach. Solutions by CVC4 were  
494 accepted as-is, in order to use it as a baseline, despite the fact that, as seen in Section 2.1,  
495 those solutions are at times overfitted and full of case-splits, but for every such case a  
496 hand-crafted gold-standard solution was also added.

497 All benchmarks were run on a Lenovo laptop with a i7 quad-core CPU @ 2.60GHz with  
498 16GB of RAM.

499 **Research questions**

500 **RQ1: Can Bester discard contradicting examples better than a naive search using**  
 501 **a state-of-the-art tool?** To test this, we examine the result of running BESTER  
 502 on the “no-solution” portion of the modified benchmark set. We run BESTER with  
 503 a 40s timeout, which is not enough for a simple enumerating synthesizer to find a  
 504 satisfying program for these tasks. We then test whether a gold-standard program for  
 505 the original benchmark was returned as the top-ranked result, and compare to the  
 506 ability of CVC4 to find the gold-standard result when run first with the full example  
 507 set and then with reduced example sets.

508 **RQ2: Can Bester rank a gold-standard result high when there is an overfitted,**  
 509 **unintended result for the example set?** To test this, we examine the “overfitted”  
 510 portion of the modified benchmark set. We still ran BESTER with a 40s timeout, but  
 511 since a fully satisfying result exists, these benchmarks terminate before the timeout.  
 512 Though BESTER will find a fully-satisfying result to the examples, it will also return  
 513 other best-effort results. We search for a gold-standard solution in the top 3 results  
 514 for each task.

515 **RQ3: Can Bester find pieces of a gold-standard solution when the task is too hard**  
 516 **for it to synthesize?** To test this, we search for pieces of gold-standard solutions in  
 517 the top results when enumerating the “hard” benchmark set. This question is further  
 518 examined in the user study in Section 6.

519 **RQ4: Does the best-effort enumeration in Bester interfere with its ability to solve**  
 520 **a simple synthesis task?** In other words, can BESTER solve the “easy” benchmark  
 521 set, returning the gold-standard solution as the top-rated result?

522 **5.1 Erroneous examples**

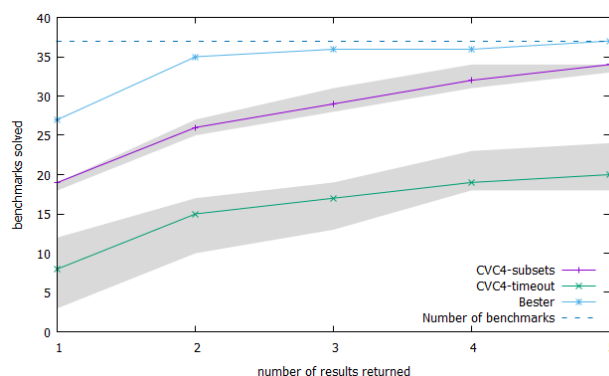
523 In RQ1 and RQ2, we wish to empirically quantify the effort of a user looking at a list of  
 524 results. That the gold-standard solution appear *somewhere* on the list of programs shown  
 525 as a result to a synthesis call is necessary but insufficient. Ideally, the user would have to  
 526 look through as few programs as possible until they find the one they are looking for—and for  
 527 confidence in the tool to be high, this should also be consistent.

528 Since CVC4 only returns one result that satisfies all examples, it will successfully synthesize  
 529 none of the modified benchmarks by construction of the benchmark set. To test RQ1 and  
 530 RQ2, we implemented a naive best-effort search using CVC4:

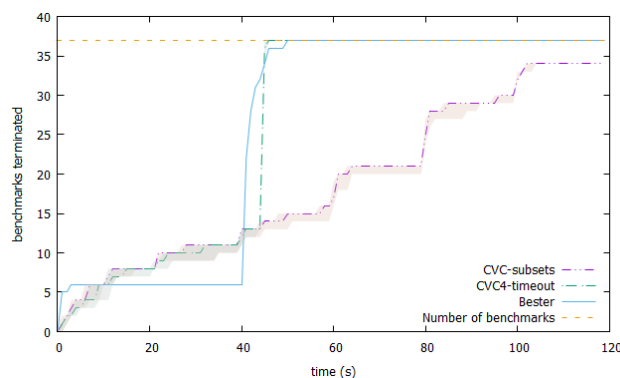
- 531 ■ CVC4-SUBSETS runs on  $\mathcal{E}$ , and then on all subsets of size  $|\mathcal{E}| - 1$  in a random order. Each  
 532 such run is done with a 20s timeout (a longer timeout would give BESTER an unfair  
 533 advantage in the measurements, and as can be seen in Figure 4b, 20s is sufficient for  
 534 CVC4 for most of the unmodified benchmarks), and results are accumulated in the order  
 535 that they are discovered and deduplicated in-order.
- 536 ■ CVC4-TIMEOUT runs as CVC4-SUBSETS, but with an additional overall timeout of 45s, in  
 537 order to be comparable to BESTER.

538 We ran the 37 modified benchmarks with BESTER, CVC4-SUBSETS, and CVC4-TIMEOUT.  
 539 Since CVC4-SUBSETS and CVC4-TIMEOUT depend on the random order of the dropped  
 540 examples, we ran each 5 times and indicate the median and variance. The results are shown  
 541 in Figure 3.

542 **RQ1: Can BESTER discard contradictions in the example set?** Out of 31 benchmarks in  
 543 the “no-solution” subset of the modified benchmarks, BESTER returned the gold-standard  
 544 solution first for 26, and the remaining 5 as the second solution. CVC4-SUBSETS returned



(a) Number of benchmarks in which the gold-standard solution was returned for a given length of result list. More benchmarks in which a gold-standard solution was found in a shorter list is better. CVC4 runs include a random component, so indicated is the median over 5 runs, with the shaded area indicating the variance.



(b) Number of benchmarks that terminate within a given length of time. This is irrespective of correctness, as the tool must first terminate for its results to be judged by the user. CVC4 runs include a random component, so indicated is the median over 5 runs, with the shaded area indicating the variance. The first plateau for BESTER indicates the “overfitted” benchmark set, where a fully-satisfying but overfitted program is found within the timeout.

■ **Figure 3** Correctness and termination times on benchmarks containing at least one erroneous example.

545 the gold-standard solution within the top 3 for only 25 of the 31 “no-solution” benchmarks  
 546 (over 5 runs, min 23, max 28), notably failing completely to synthesize a specification with  
 547 more than one erroneous example, of which “no-solution” contains two. In addition, it only  
 548 returned the gold-standard solution first for 17 of the benchmarks (min 16, max 18), with  
 549 some gold-standard solutions being as low as fifth. Finally, CVC-TIMEOUT fails to return a  
 550 gold-standard solution in the top 5 for 15 of the 31 benchmarks (min 13, max 18), and only  
 551 returns the gold-standard solution first for 7 of them (min 2, max 9).

552 We therefore conclude that BESTER is effective at discarding contradictions from the  
 specification and returning a desirable program to the user. Additionally, we conclude  
 that our efficient best-effort implementation is more efficient than a naive approach  
 using a state of the art synthesizer.

553 **RQ2:** Can BESTER return a useful solution despite an overfitted program matching the  
 554 examples? Out of the remaining 6 “overfitted” modified benchmarks, BESTER shows 5 in the  
 555 top three results and 4 in the top 2, exactly the same as CVC4-SUBSETS (min 4, max 6 and  
 556 min 3, max 4, respectively). CVC4-TIMEOUT had 4 in the top three results (min 3, max 5)



557 and 3 in the top two (min 2, max 4).

558 We can also see the “overfitted” benchmarks in Figure 3b, as the first plateau between 3  
559 and 40 seconds: overfitted programs are found quickly, and other program options collected  
560 along the way are also shown to the user, as opposed to enumerating a benchmark from  
561 “no-solution”, which will continue until the timeout.

562 We conclude that BESTER performs as well as CVC4-SUBSETS and CVC4-TIMEOUT at  
ranking the gold-standard solution in the top 3 when an overfitted solution exists. This  
is done more efficiently than a naive solution implemented with CVC4, which still pays  
the overhead of having to perform multiple runs.

## 563 5.2 Partially solving hard benchmarks

564 In RQ3, we examine the results of BESTER on the “hard” set of benchmarks, which are  
565 benchmarks that a simple enumerating OE-reduction synthesizer cannot complete within 40s.  
566 BESTER also runs with a timeout of 40s, but returns any best-effort results it finds. None  
567 of the results returned will be a gold-standard solution, but they may be part of a path to  
568 a solution. Therefore, to answer RQ3, we try to quantify how much of each of the results  
569 returned by BESTER can be used to construct a solution.

570 In order to do that, we must first define the way we measure this similarity.

### 571 Tree similarity

572 In order to judge how much of a result returned by BESTER is relevant to the user, we use  
573 a similarity metric between trees on the ASTs of the BESTER result and the gold-standard  
574 solution. This metric essentially counts what non-trivial parts of the code can be copied out  
575 verbatim.

576 When computing  $s(p_1, p_2)$ , we look for maximal sub-expressions (or subtrees)  $x$  within  
577  $p_1$  (denotes  $x \in p_1$ ) that are also included in  $p_2$ . For each such  $x$ , if  $height(x) > 0$  (i.e.,  $x$   
578 is not a leaf node) we count  $terms(x)$ . Additionally, we reward the same term for using  
579 some identical children even if not *all* children are identical. For example, if there exist  
580 two trees,  $t(x, y) \in p_1$  and  $t(x, z) \in p_2$  (notice that  $t$  is the same node type and  $x$  is in the  
581 same location) we count the root  $t$  in addition to  $terms(x)$ , i.e., add 1 to the accumulated  
582 similarity.

583 Equivalent programs that result in structurally different trees (e.g., `(str.++ "be seeing"`  
584 `(str.++ " " "you"))` vs. `(str.++ (str.++ "be seeing" " ") "you")`) were handled manually  
585 by first performing equivalence-preserving tree transformations on the gold standard and  
586 then computing the similarity.

587 Other similarity metrics were originally considered. Program repair projects often employ  
588 distance metrics between programs to choose between several possible repairs. Distance  
589 metrics for structured objects such as DIFFX [1] for XMLs were applied to ASTs, and  
590 application-specific ones were crafted [15, 57]. However, the fragment mapping employed by  
591 such distances is more useful for describing insertion and deletion of code (e.g., wrapping a  
592 part of the tree in a conditional, removing a statement), whereas we are interested in pieces  
593 of code that can be used without modification.

### 594 Usable parts of best-effort solutions

595 We ran BESTER on the 16 benchmarks in the “hard” set. The results are shown in Table 1.

596 **RQ3:** *Can BESTER return a useful best-effort solution for tasks that it cannot solve within*  
597 *the timeout?* On average, BESTER results discover over 40% of the gold-standard solution to a

benchmark	gold standard			top BESTER solution				closest solution to GS				
	# GS	avg h	avg t	h	t	sim	% best GS	rank	h	t	sim	% best GS
11604909	3	3.7	15.0	2	12	8	62%	1	2	12	8	62%
30732554	1	3.0	12.0	0	1	0	0%	1	0	1	0	0%
38871714	2	6.0	19.0	2	7	7	37%	1	2	7	7	37%
39060015	2	11.0	72.0	2	6	0	0%	2	2	7	15	45%
41503046	3	8.0	64.7	2	7	11	7%	1	2	7	11	7%
43606446	2	5.5	24.5	3	16	12	38%	1	3	16	12	38%
44789427	3	5.7	40.3	2	7	15	21%	2	2	11	16	73%
bikes	2	4.0	16.5	3	14	13	50%	3	3	14	20	77%
count-total-words	1	5.0	35.0	3	14	22	63%	3	3	20	23	66%
exceljet2	1	7.0	43.0	2	11	14	33%	1	2	11	14	33%
stackoverflow1	1	3.0	16.0	2	9	9	56%	1	2	9	9	56%
stackoverflow2	1	6.0	28.0	3	20	24	86%	1	3	20	24	86%
stackoverflow3	1	4.0	12.0	2	6	0	0%	1	2	6	0	0%
strip-html	1	4.0	15.0	-	-	-	-	-	-	-	-	-
univ_2_short	1	4.0	20.0	2	7	7	35%	1	2	7	7	35%
univ_3_short	1	4.0	14.0	0	1	0	0%	1	0	1	0	0%

■ **Table 1** Portions of the gold-standard solutions discovered by BESTER for the tasks in the “hard” set. The first set of columns is information on the gold standard solutions available for a task: number and average size. The second set shows the program BESTER ranked first: size, its similarity to the most similar gold-standard solution, and what percentage of the terms in the gold-standard solution is covered ( $sim(p,gs)/terms(gs)$ ). For the closest solution to a gold-standard solution, the rank of the program in BESTER’s list is also indicated. t denotes terms, h denotes height (this is zero-based), sim denotes the similarity to most similar gold-standard solution.

598 task (or the most similar one, if there is more than one), or over 11 terms. When considering  
 599 only the programs ranked first by BESTER, 32% of the gold-standard solution is discovered  
 600 with an average of almost 9.5 terms. In 3 of the benchmarks, the entire top-ranking BESTER  
 601 result was a sub-expression of the solution to the task.

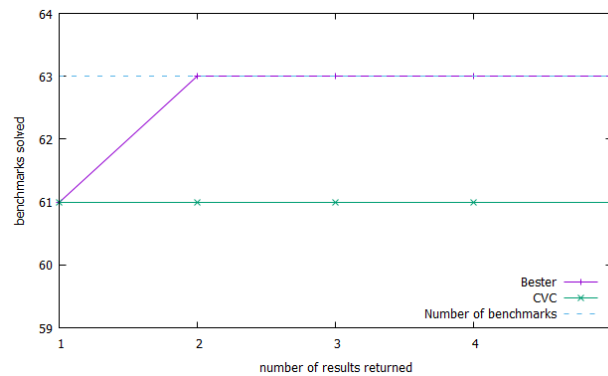
602 Notice that in some of the tasks (e.g., STACKOVERFLOW2) the similarity between the  
 603 BESTER result and its nearest gold-standard solution is greater than the number of terms in  
 604 the BESTER result. This is because an expression in the BESTER result can repeat multiple  
 605 times in the gold-standard solution.

606 In one of the 16 benchmarks, BESTER did not find any program that satisfies at least one  
 607 example, and so returned no programs. In 3 additional benchmarks, none of the programs  
 608 returned had any non-trivial subtree in common with a gold-standard solution.

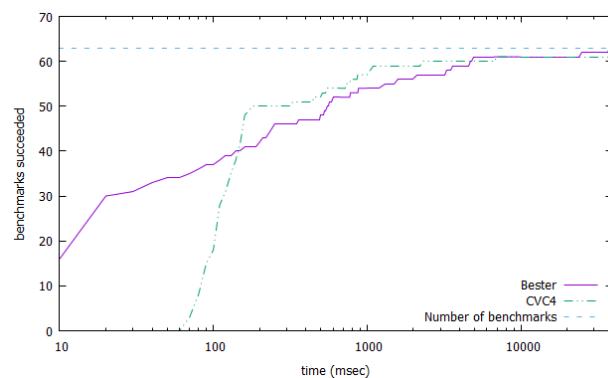
609 Overall, we conclude that BESTER generates results that can advance the user toward a solution even when they do not fully satisfy the specification. This will be further demonstrated in Section 6. Even though in some of the benchmarks none of the results had any usable components, these are still a minority (overall a quarter of the benchmarks) and the high similarity of those that did succeed indicates the approach can be of great use to a user.

### 610 5.3 Solving the original easy benchmarks

611 Since BESTER ranks its results, taking into account but not relying solely on the number of  
 612 examples satisfied (see Section 4), we must verify that the solutions to the original, unmodified  
 613 benchmarks that can be solved by the simple OE synthesizer are still found.



(a) Number of benchmarks in which the gold-standard solution was returned for a given length of result list. CVC4 only returns a single result.



(b) Number of benchmarks that terminate within a given length of time, graph is logscale.

■ **Figure 4** Correctness and time to solution on “easy” benchmarks. CVC4, which was part of the baseline for correct results, is correct every task that terminates within the 40s timeout. CVC4 is faster, but the difference is not extreme.

614 To test RQ4, we ran BESTER and CVC4 with a 40s timeout on the unmodified “easy” set  
615 of benchmarks. The results are in Figure 4.

616 **RQ4:** *Can BESTER return the correct result for unmodified “easy” benchmarks?* BESTER  
617 succeeds in returning a correct solution that is ranked first for 61 out of 63 of the benchmarks  
618 in the “easy” set, on par with the performance of CVC4. (Since CVC4 was used in the  
619 creation of the gold standard, it succeeds on every benchmark it terminates on within the  
620 40s timeout.)

621 In the remaining two benchmarks, the gold standard solution is ranked second. In both  
622 of these benchmarks, the desired outputs are a substring of `arg0`, the input variable. Both  
623 also contain multiple examples where the input is unchanged. For both of these benchmarks,  
624 BESTER ranks the program `arg0` before the target program, since it satisfies some of the  
625 examples and is very close to the correct output in the others, uses all the variables, and is  
626 very simple. This is rare, and when presented to a user, as in Figure 1a, the program would  
627 be accompanied by the number of benchmarks it solves, and we believe it will be easy for  
628 users to discard.

629 Additionally, BESTER is not considerably slower than CVC4 on the benchmarks in “easy”.

We conclude that implementing the best-effort enumeration in BESTER does not harm its correctness on benchmarks that contain no error or contradiction, and that its efficiency in such cases is not much worse than a state of the art synthesizer optimized for competitions.

## 6 An Exploratory User Study

In this section we detail the results of a small exploratory study in which 8 users were asked to use BESTER to perform two tasks each. Tasks were selected from the benchmark suite presented in Section 5, from the “hard” set of benchmarks, i.e., benchmarks that could not be solved within the timeout by a simple enumerating synthesizer. Notice that these are not modified tasks, i.e., they are identical to their version in the Euphony benchmark set from which they both originated. After completing the tasks, we asked each user to answer a set of questions in a brief interview.

### Experiment setup

8 graduate students participated in the study. Users were presented with a brief task description (as it appears in a comment in the benchmark file), the examples in the benchmark, and the grammar at their disposal. As shown in Section 2.1, the semantics of some grammar elements can be misleading, particularly in edge cases.

Participants used a REPL for the target SYGUS language that is initialized with the limited grammar and the example set for the task. For each program entered, the REPL prints the output for every input in the example set. Satisfied examples (matching the example’s expected output) are indicated in green. Screenshots of the REPL are shown in Figure 1. Participants could edit the program on their own or, at any point, call the synthesizer to find a program that would satisfy the examples. The BESTER synthesizer runs either until a timeout of 40s or until interrupted by the user (“press any key” in Figure 1b). While synthesis ran, a number showing the maximum number of examples satisfied was shown and updated when new programs were found. The top 5 programs found by the synthesizer are presented to the user, and can be executed or copied. Participants could call the synthesizer multiple times in the course of one task, as a longer wait could possibly yield more results.

A task was concluded when a participant said they solved the task, when they gave up on the task, or when 20 minutes had elapsed.

Users were told that the tasks are underspecified, and they may resolve any ambiguity as they see fit. Correctness was judged according to semantic equivalence to one of the gold standard solutions for Section 5.

After performing the tasks, users were given a brief structured interview with questions about their use of the synthesizer and the helpfulness of the results. Each participant was paid \$10.

### Study tasks

The two tasks given to the participants shared a SYGUS grammar, differing only in the available string literals.

**Task 1:** “stackoverflow1” in Table 1. Its comment in the benchmark file, provided to participants, was “function to replace substring”.

	arg0	expected result
670	Examples: "Trucking Inc."	"Trucking"
	"New Truck Inc"	"New Truck"
	"ABV Trucking Inc, LLC"	"ABV Trucking"

671 The available string literals were: "", " ", "Inc", ".", ",", and "LLC".

672 **Task 2:** "41503046" in Table 1. Its comment in the task file was "find string in substring  
673 with lookup".

	arg0	expected result
	"Polygonum amphibium"	"Polygonum"
	"Hippuris vulgaris"	"Hippuris"
674	Examples: "Lysimachia vulgaris"	"Lysimachia"
	"Juncus bulbosus ssp. bulbosus"	"Juncus bulbosus"
	"Lycopus europaeus ssp. europaeus"	"Lycopus europaeus"
	"Nymphaea alba"	"Nymphaea"

675 The available string literals were: "", " ", and "ssp."

## 676 Research questions

677 In order to find out whether the *best-effort* paradigm can be useful to programmers, we  
678 attempt to answer the following questions:

679 **RQ1:** Did users apply any part of the results from BESTER to their solution?

680 **RQ2:** Did users find the results from BESTER helpful even though they do not satisfy every  
681 example?

## 682 6.1 Observed behavior

683 Participants completed task 1 in an average of 9.56 minutes and task 2 in an average of 11.35  
684 minutes. The fastest solution was programmed in just under 5 minutes.

685 Of 8 users performing two tasks each, 7 successfully completed both tasks. One user  
686 failed to finish the first task within the 20 minute bound and successfully finished the second  
687 task. In addition, one user finished the second task with an incorrect result, and, as they  
688 were not satisfied with it and had time left, continued to rewrite it until they reached a  
689 correct result.

690 In 15 of the 16 task sessions, the users called the synthesizer at some point during the  
691 session. In task 1, 3 of the users ran the synthesizer a second time in the course of the session.  
692 In task 2, 2 of the users did so, and one ran the synthesizer a third time. One user performed  
693 task 1 without running the synthesizer at all.

694 Users waited for the synthesizer an average of 17.5s per session while working on task 1  
695 and 27.6s per session while working on task 2, or an average of 14s per individual run of the  
696 synthesizer for task 1 and an average of 18.4s for task 2. Only twice did users allow their  
697 synthesis request to run until the 40s timeout, both in the course of solving task 2.

698 7 of the 8 participants executed the top synthesis result once the synthesizer terminated.  
699 Only one user executed any result other than the top result—and they executed all results.  
700 6 users later returned to an executed synthesis result using the REPL history and continued  
701 to edit it from there.

702 6 of the users used the mouse to highlight and copy a synthesized expression and paste it  
703 into their code. Two users also copied parts of a synthesized expression, but for the most  
704 part, the synthesis results that were copied by users were used in their entirety and placed  
705 within larger expressions.

706 Task 1 has two possible modes of solution: one using `str.substr` to slice the string up  
707 to the occurrence of "Inc" and using `str.replace` to replace undesirable substrings with "".  
708 Four users followed the synthesizer's lead in solving the task with `str.replace`, and another  
709 user attempted this and abandoned the direction.

710 Of the 8 users, 5 ran the synthesizer immediately upon being given task 1 (of the 3 who  
711 did not, one did not run the synthesizer at all), and 7 ran it immediately upon being given  
712 task 2.

713 Many of the participants struggled with the behavior of the `str.indexOf` function which  
714 returns the index of a substring within a string. Unlike the simplified version included in the  
715 grammar in Figure 2a, the function takes an integer parameter which indicates at what index  
716 the search for the substring should begin. Many of the users assumed the index parameter to  
717 indicate which occurrence of the string should be returned. In the solution of task 2, users  
718 spent some time trying to get the second occurrence of " " under this assumption.

## 719 6.2 Interviews

720 In the interview conducted after the tasks were concluded, participants were asked about  
721 their decision to call the synthesizer (and to call it again in the course of the session, if  
722 they did so), about how they decided how long to wait for the synthesizer, and about the  
723 helpfulness of the results.

### 724 Calling the synthesizer

725 Several users explained their call to synthesis as a way to search for a solution they were not  
726 seeing, or in hopes it will simply solve the task for them (or, in the case of one user, "just to  
727 see what it can do"). Some also recognized, particularly for task 2, that there may be at  
728 least a subproblem that can be solved by the synthesizer, providing them with "a start on  
729 the solution" or "a piece that can be reused".

730 However, many of the users explained their call to synthesis as a way to help them  
731 understand the problem: either by seeing if there was a generalization of the examples they  
732 were not considering, or to get a confirmation of their understanding, "make sure the model  
733 in [their] head was correct".

734 The user who performed task 1 without synthesis said they did not think there exists a  
735 simpler way to perform the task than the one they had in mind, so there was no need for  
736 synthesis.

737 Finally, many of the users explained that synthesized code was, to them, a good source of  
738 example programs on the inputs. Synthesized code gave them examples of *a*) the language  
739 syntax and useful available functions, *b*) the semantics of the functions, and the order of  
740 the arguments, *c*) function composition, and how different functions interact, and *d*) help  
741 dealing with what one of the users called "an unnatural collection of primitives".

### 742 Waiting for the synthesizer

743 Most users who ran the synthesizer immediately at the start of the task attested that it  
744 seemed to them a good use of time to let it run as they were reading the task — it might  
745 find something and save them the effort. One user ran the synthesizer again (and to timeout)  
746 while they were thinking through a problem they had encountered, just in case.

747 Users could stop the synthesizer at any time. Three of the users said they used the  
748 printout of how many examples were solved by the best discovered program as an indication of

749 when to stop: “[as long as] it made some progress, it was fine”. When the number plateaued,  
750 they “figured it solved part of the problem, but the rest isn’t easy.”

751 Frustration was also a deciding factor in willingness to wait. Users who were not having  
752 a hard time with the tasks and simply wanted some reference, terminated the synthesizer  
753 very quickly, and they just wanted to see the first results rather than be slowed down by  
754 waiting. Users who were more frustrated, especially those who entered task 2 frustrated from  
755 task 1, expressed being more willing to wait. The user who failed to finish task 1 and ran  
756 the synthesizer to timeout (40s) in task 2 said, “I really struggled, so even if the timeout  
757 was 10 minutes, it’s worth it.”

758 Only two of the 8 users explicitly named impatience as the criterion for deciding how  
759 long to wait for the synthesizer.

760 Half (4) the users re-ran the synthesizer within the course of the same task for one of  
761 the two tasks. All said it was in hopes that waiting longer would produce more or better  
762 results. One did so because they lost their train of thought and wanted to start over from a  
763 synthesized solution in order to remember what they were trying to do, and had forgotten  
764 they can call up the solutions from the last run of the synthesizer. This user also stated  
765 that, as they were struggling a bit, they were now more willing to wait for a result. Two  
766 users stated wanting to utilize time when they had stopped to think about what to do next,  
767 in case better solutions would be found. (One user who did not run it a second time said  
768 that “in hindsight, letting it run while I was thinking would have been good.”) One user said  
769 they were curious as to whether there was a random component that would lead to different  
770 results.

### 771 Helpfulness of the results

772 All participants stated the synthesized results were helpful to them in some way.

773 **Getting to a solution:** In each of the tasks, the synthesizer returned a different kind  
774 of a sub-solution. In task 1, it needed to be wrapped in more function applications to solve  
775 more cases, whereas task 2 required a case-split and the synthesizer returned a solution to  
776 one of the two cases. Some users viewed one as far more helpful than the other, though which  
777 one was not a constant. Some treated the solution to task 1 as “nearly solved the problem”,  
778 whereas others saw the solution to task 1 as less helpful but the solution to task 2 as giving  
779 them the subprogram that they wanted, where “I could just steal that as a subcomponent”.

780 **Comprehension of the language:** Participants who used the synthesizer to understand  
781 the language said synthesized results gave them “phrases” for later use and what constants  
782 were available; “here is some code, here’s what it does.” (In task 2, when they got used to  
783 the language, it was less helpful). Those who did not trust themselves with the language  
784 trusted synthesized code.

785 **Comprehension of the task:** Users also attested that synthesized results helped them  
786 better understand the task itself and in what way the examples generalized. This was  
787 particularly true in the second task which contained a case-split. Users said the result of the  
788 synthesizer classified the examples for them into the two cases of the split, or as one user said,  
789 “once I saw the response from the synthesizer, I knew exactly what the correct answer was.”

## 790 6.3 Discussion

791 We return to our research questions:

792 **RQ1:** *Did users use BESTER results in their code?* Participants of our study used both  
793 entire BESTER results and subprograms of them in their solution code. In addition, in task 1,

794 several participants let the synthesizer direct the algorithm of their solution. We therefore  
795 answer this question in the **affirmative**.

796 **RQ2:** *Did users find the results of BESTER helpful?* Participants of our study listed  
797 different ways in which the results of BESTER were helpful to them, including (but not limited  
798 to) finding code that solves a subproblem. Synthesizer results were also widely used as a  
799 comprehension tool by the users. We therefore answer this question in the **affirmative**.

## 800 6.4 Threats to validity

801 Finally, we briefly discuss the threats to the validity of our conclusions from the study.

802 **Number of participants and number of tasks:** The study was conducted on 8  
803 participants, performing only two tasks each, which is not enough to make any statistically  
804 significant claims. We therefore try to steer away from such conclusions, and instead observe  
805 and report usage patterns that occurred throughout user sessions.

806 **Selection of programming language:** While using the SYGUS language can be seen as  
807 an advantage of the study, mimicking a situation where users are not the most familiar with  
808 the language or API they are using, and therefore need the help of a synthesizer, it is also  
809 not the easiest programming language to read or write, and includes nontrivial semantics for  
810 some of its functions (as demonstrated both in Section 2.1 and in this section). This may lead  
811 to different results than a synthesizer for a programming language users are more comfortable  
812 reading and writing. All the participants in the study were familiar with the S-expression  
813 syntax and had some experience in using it, mitigating some of the comprehension difficulty  
814 if not the problematic semantics.

815 **Homogeneous participants:** Since students were recruited from a single department  
816 in a single institution, there is great similarity in their knowledge and ability. This may have  
817 resulted in similar behaviors in the course of the study.

818 **Inability to specify the synthesizer:** The BESTER implementation used in the study  
819 was not fully-equipped for an iterative and interactive workflow, and users could not control  
820 the specifications the synthesizer attempted to solve. This also means users did not spend  
821 time on (or have a learning curve in) entering specifications or deciding what they should be.  
822 Within such a larger workflow, the observed behaviors may be different. However, we have  
823 tried to only draw conclusions about the usefulness of the results of a synthesizer iteration,  
824 rather than on the interactive incorporation of synthesis in the development workflow.

## 825 7 Related work

826 **Syntax-guided synthesis** [3] is the domain of program synthesis where the target program  
827 is derived from a set of syntax rules. [30, 55, 18, 56] all fall within this scope. FLASHFILL  
828 and FLASHMETA [23, 42] are tools for automating string transformations and data wrangling  
829 tasks, whose DSL design centers the delicate balance between an expressive grammar, which  
830 is needed to find a solution, and a tractable enumeration. Padhi et al. [38] raise the issue of  
831 the overfitting of an over-expressive grammar, leading to programs such as the one shown in  
832 Section 2.1.

833 **The SyGuS competition** [5, 4] is held every year and allows solvers and synthesizers  
834 to compete for both performance and correctness on a large selection of benchmarks. The  
835 competition introduced a PBE track in 2016, and now has two PBE tracks, one for string tasks  
836 and one for bit-vector tasks. Both CVC4 [45] and EUSolver [6] have won the competition in  
837 the past.



838 **Programming by Example** is a popular technique in program synthesis that leverages  
839 either user-provided input-outputs [56, 36, 21, 25, 23, 24, 42, 58] or tests [20]. Most PBE  
840 techniques target exact specifications and do not handle **noise** in user input. Some notable  
841 exceptions are FLASHFILL [23] and RULESYNTH [48], as well as Bayesian and neural program  
842 induction techniques [16, 17, 53]. None of these approaches, however, compute results for all  
843 subsets of examples, or deal with timeouts.

844 **Ranking** and returning multiple results are two common approach to handing ambiguous  
845 specifications in program synthesis; the two often—but not always—go hand-in-hand. The  
846 FLASHX tool family [23, 42] uses a ranking function to select a single, most likely program  
847 among all the programs that satisfy all user-provided examples. This line of work has  
848 explored both hand-crafted [23] and learned [47] ranking functions. Recent work on guiding  
849 synthesis using learned probabilistic models [33] can also be seen as applying a learned  
850 ranking, but during synthesis rather than at the end. Our ranking function for BESTER is  
851 hand-crafted, but is different from existing work in that it incorporates semantic features of  
852 programs in addition to syntactic ones, such as the number of examples satisfied, and the  
853 distance between the expected and actual outputs. Recent work on synthesizing lenses [35]  
854 proposed a novel approach to semantic ranking based on information theory. In the future  
855 we would like to explore whether best-effort synthesis can benefit from a more sophisticated  
856 ranking function along these lines. Unlike PBE tools, which use ranking to select a single  
857 result, code completion tools [28, 43] typically present a ranked list of results to the user,  
858 and most commonly rely on learned statistical models and syntactic features.

859 **Observational equivalence** Many enumerating synthesizers apply equivalence reduc-  
860 tions as a form of pruning the program space [28, 36, 22, 21, 49]. Observational equiva-  
861 lence [2, 55], as a more aggressive and therefore more optimizing form of equivalence, is used  
862 in many bottom-up synthesizers [56, 6, 51, 41].

863 EUSOLVER [6] specializes in solving benchmarks that require case-splitting by performing  
864 an OE-reduced enumeration searching for two subprograms that together cover the examples  
865 and a condition to decide between them. The enumeration performed by EUSOLVER is similar  
866 to that of BESTER in that it is an enumeration over all the examples that also considers  
867 subsets of the examples, but only the first program covering a specific subset of the examples  
868 is used within the (single) result program, whereas BESTER ranks all such programs and  
869 returns the highest ranking ones even if several of them cover the same subset of the examples.

870 **Interaction models for program synthesis** are a recent field of research, which has  
871 taken two main directions: Modifying specification mechanisms and output formats [40, 13]  
872 to make synthesis easier to use and better targeted to specific populations of users. Iterative  
873 program synthesis [32, 39, 7] focuses on allowing the user to refine the specification while  
874 running the synthesizer after each such refinement, essentially making explicit and improving  
875 upon what has been the implicit assumption of all synthesis tools. BESTER is currently  
876 situated well within the first direction, but we believe it will also aid the users greatly in an  
877 iterative setting.

878 **MaxSAT** [10] and MaxSMT [9] are the formulation of the satisfiability problem in which  
879 certain clauses are marked as *hard constraints* and others as *soft constraints*, and the solver  
880 attempts to find an assignment that satisfies all hard constraints while maximizing the  
881 number of soft constraints satisfied. Viewing the world through this terminology, we see  
882 that previous work has viewed user-provided inputs as hard constraints, and even in work  
883 where other soft constraints are available to the user [14], examples are still considered hard  
884 constraints. In BESTER, all examples are soft constraints and a ranking function is being  
885 maximized likening our paradigm to weighted MaxSAT. In Section 4 we suggest a case where

886 there could be additional weights between the specifications.

887 **Opportunistic programming** [12, 11] is the programming paradigm in which composite  
 888 programming tasks are solved by hunting for and joining pieces of existing code from other  
 889 sources. Projects such as EXAMPLESTACK [59] are intended to make the process of importing  
 890 found code easier. The BESTER user study in Section 6 demonstrates synthesis as another  
 891 method that can provide *pieces* of the solution to the programmer.

## 892 **8 Conclusion**

893 We proposed a new program synthesis paradigm we call *best-effort program synthesis*, where  
 894 the synthesizer returns a ranked list of programs that satisfy some part of the specification,  
 895 rather than just one program that satisfies all of it or no program at all.

896 This paradigm is implemented in a *best-effort enumeration*, a new synthesis algorithm that  
 897 extends a bottom-up enumeration with observational equivalence, and is able to accumulate  
 898 multiple partially-valid results with minimal overhead. We implemented this algorithm in a  
 899 tool called BESTER, and evaluated it on 79 synthesis benchmarks from the SYGUS competition  
 900 and the EUPHONY benchmark suite.

901 Our empirical evaluation showed that best-effort enumeration is more efficient and returns  
 902 better results than a naive approach to best-effort program synthesis, and that BESTER  
 903 returned useful results even when the specification is flawed or too hard: *i*) for specifications  
 904 containing an erroneous example, the top three BESTER results contained the correct solution,  
 905 and *ii*) for most hard benchmarks, the top three results contained non-trivial *fragments*  
 906 of the correct solution. Our user study showed that users apply partially-valid results and *parts*  
 907 of those results to their code. Additionally, we observed that programmers use the output of  
 908 the synthesizer for comprehension and not only as a possible part of their solution.

## 909 **References**

- 
- 910 1 Raihan Al-Ekram, Archana Adma, and Olga Baysal. diffx: an algorithm to detect changes in  
 911 multi-version xml documents. In *Proceedings of the 2005 conference of the Centre for Advanced  
 912 Studies on Collaborative research*, pages 1–11. IBM Press, 2005.
  - 913 2 Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In  
 914 *International Conference on Computer Aided Verification*, pages 934–950. Springer, 2013.
  - 915 3 Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman,  
 916 Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa.  
 917 Syntax-guided synthesis. *Dependable Software Systems Engineering*, 40:1–25, 2015.
  - 918 4 Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2017:  
 919 Results and analysis.
  - 920 5 Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2016:  
 921 Results and analysis. *arXiv preprint arXiv:1611.07627*, 2016.
  - 922 6 Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program  
 923 synthesis via divide and conquer. In *International Conference on Tools and Algorithms for the  
 924 Construction and Analysis of Systems*, pages 319–336. Springer, 2017.
  - 925 7 Shengwei An, Rishabh Singh, Sasa Misailovic, and Roopsha Samanta. Augmented example-  
 926 based synthesis using relational perturbation properties. *Proceedings of the ACM on Program-  
 927 ming Languages*, 4(POPL):56, 2019.
  - 928 8 Pavol Bielik, Veselin Raychev, and Martin Vechev. Phog: Probabilistic model for code. In  
 929 Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International  
 930 Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*,  
 931 pages 2933–2942, New York, New York, USA, 20–22 Jun 2016. PMLR. URL: [http://  
 932 proceedings.mlr.press/v48/bielik16.html](http://proceedings.mlr.press/v48/bielik16.html).

- 933 9 Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein.  $\nu z$ -an optimizing smt solver. In  
934 *International Conference on Tools and Algorithms for the Construction and Analysis of*  
935 *Systems*, pages 194–199. Springer, 2015.
- 936 10 Brian Borchers and Judith Furman. A two-phase exact algorithm for max-sat and weighted  
937 max-sat problems. *Journal of Combinatorial Optimization*, 2(4):299–306, 1998.
- 938 11 Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. Two  
939 studies of opportunistic programming: interleaving web foraging, learning, and writing code.  
940 In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages  
941 1589–1598. ACM, 2009.
- 942 12 Joel Brandt, Philip J Guo, Joel Lewenstein, and Scott R Klemmer. Opportunistic programming:  
943 How rapid ideation and prototyping occur in practice. In *Proceedings of the 4th international*  
944 *workshop on End-user software engineering*, pages 1–5. ACM, 2008.
- 945 13 Sarah Chasins. *Democratizing Web Automation: Programming for Social Scientists and Other*  
946 *Domain Experts*. PhD thesis, EECS Department, University of California, Berkeley, Oct 2019.  
947 URL: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-139.html>.
- 948 14 Yanju Chen, Ruben Martins, and Yu Feng. Maximal multi-layer specification synthesis. In  
949 *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference*  
950 *and Symposium on the Foundations of Software Engineering*, pages 602–612, 2019.
- 951 15 Loris D’Antoni, Roopsha Samanta, and Rishabh Singh. Qclose: Program repair with quantitative  
952 objectives. In *Computer Aided Verification - 28th International Conference, CAV 2016,*  
953 *Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 383–401, 2016. doi:  
954 10.1007/978-3-319-41540-6\_21.
- 955 16 Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed,  
956 and Pushmeet Kohli. Robustfill: Neural program learning under noisy I/O. In *Proceedings of*  
957 *the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia,*  
958 *6-11 August 2017*, pages 990–998, 2017.
- 959 17 Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer  
960 graphics programs from hand-drawn images. In S. Bengio, H. Wallach, H. Larochelle, K. Grau-  
961 man, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing*  
962 *Systems 31*, pages 6059–6068. Curran Associates, Inc., 2018. URL: [http://papers.nips.cc/](http://papers.nips.cc/paper/7845-learning-to-infer-graphics-programs-from-hand-drawn-images.pdf)  
963 [paper/7845-learning-to-infer-graphics-programs-from-hand-drawn-images.pdf](http://papers.nips.cc/paper/7845-learning-to-infer-graphics-programs-from-hand-drawn-images.pdf).
- 964 18 Azadeh Farzan and Victor Nicolet. Modular divide-and-conquer parallelization of nested loops.  
965 In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design*  
966 *and Implementation, PLDI 2019*, pages 610–624, New York, NY, USA, 2019. ACM. URL:  
967 <http://doi.acm.org/10.1145/3314221.3314612>, doi:10.1145/3314221.3314612.
- 968 19 Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-  
969 based synthesis of table consolidation and transformation tasks from examples. In *Proceedings*  
970 *of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation,*  
971 *PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 422–436, 2017.
- 972 20 Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W Reps. Component-based  
973 synthesis for complex apis. *ACM SIGPLAN Notices*, 52(1):599–612, 2017.
- 974 21 John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations  
975 from input-output examples. In *ACM SIGPLAN Notices*, volume 50, pages 229–239. ACM,  
976 2015.
- 977 22 Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed  
978 synthesis: A type-theoretic interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-*  
979 *SIGACT Symposium on Principles of Programming Languages, POPL ’16*, pages 802–815,  
980 New York, NY, USA, 2016. ACM. URL: <http://doi.acm.org/10.1145/2837614.2837629>,  
981 doi:10.1145/2837614.2837629.
- 982 23 Sumit Gulwani. Automating string processing in spreadsheets using input-output examples.  
983 In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of*

- 984 *Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM. URL:  
985 <http://doi.acm.org/10.1145/1926385.1926423>, doi:10.1145/1926385.1926423.
- 986 24 Sumit Gulwani. Synthesis from examples: Interaction models and algorithms. In *Symbolic and*  
987 *Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium*  
988 *on*, pages 8–14. IEEE, 2012.
- 989 25 Sumit Gulwani. Programming by examples (and its applications in data wrangling). In Javier  
990 Esparza, Orna Grumberg, and Salomon Sickert, editors, *Verification and Synthesis of Correct*  
991 *and Secure Systems*. IOS Press, 2016.
- 992 26 Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of  
993 loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming*  
994 *Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages  
995 62–73, 2011. URL: <http://doi.acm.org/10.1145/1993498.1993506>, doi:10.1145/1993498.  
996 1993506.
- 997 27 Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and  
998 Nadia Polikarpova. Program synthesis by type-guided abstraction refinement. In *Principles*  
999 *of programming languages*, page to appear, 2020.
- 1000 28 Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using  
1001 types and weights. In *ACM SIGPLAN Notices*, volume 48, pages 27–38. ACM, 2013.
- 1002 29 Jeevana Priya Inala and Rishabh Singh. Webrelate: integrating web data with spreadsheets  
1003 using examples. *PACMPL*, 2(POPL):2:1–2:28, 2018. doi:10.1145/3158090.
- 1004 30 Shachar Itzhaky, Rohit Singh, Armando Solar-Lezama, Kuat Yessenov, Yongquan Lu, Charles  
1005 Leiserson, and Rezaul Chowdhury. Deriving divide-and-conquer dynamic programming al-  
1006 gorithms using solver-aided transformations. In *Proceedings of the 2016 ACM SIGPLAN*  
1007 *International Conference on Object-Oriented Programming, Systems, Languages, and Applica-*  
1008 *tions*, pages 145–164. ACM, 2016.
- 1009 31 Vu Le and Sumit Gulwani. FlashExtract: a framework for data extraction by examples. In  
1010 Michael F. P. O’Boyle and Keshav Pingali, editors, *Proceedings of the 35th Conference on*  
1011 *Programming Language Design and Implementation*, page 55. ACM, 2014. doi:10.1145/  
1012 2594291.2594333.
- 1013 32 Vu Le, Daniel Perelman, Oleksandr Polozov, Mohammad Raza, Abhishek Udupa, and Sumit  
1014 Gulwani. Interactive program synthesis. *CoRR*, abs/1703.03539, 2017. URL: <http://arxiv.org/abs/1703.03539>, arXiv:1703.03539.
- 1015 33 Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program  
1016 synthesis using learned probabilistic models. In *ACM SIGPLAN Notices*, volume 53, pages  
1017 436–449. ACM, 2018.
- 1018 34 Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals.  
1019 In *Soviet physics doklady*, volume 10, pages 707–710, 1966.
- 1020 35 Anders Miltner, Solomon Maina, Kathleen Fisher, Benjamin C. Pierce, David Walker, and  
1021 Steve Zdancewic. Synthesizing symmetric lenses. *Proc. ACM Program. Lang.*, 3(ICFP), July  
1022 2019. doi:10.1145/3341699.
- 1023 36 Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In  
1024 *ACM SIGPLAN Notices*, volume 50, pages 619–630. ACM, 2015.
- 1025 37 Antti Oulasvirta and Pertti Saariluoma. Surviving task interruptions: Investigating the  
1026 implications of long-term working memory theory. *International Journal of Human-Computer*  
1027 *Studies*, 64(10):941–961, 2006.
- 1028 38 Saswat Padhi, Todd D. Millstein, Aditya V. Nori, and Rahul Sharma. Overfitting in synthesis:  
1029 Theory and practice. In *Computer Aided Verification - 31st International Conference, CAV*  
1030 *2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, pages 315–334, 2019.  
1031 doi:10.1007/978-3-030-25540-4\_17.
- 1032 39 Hila Peleg, Shachar Itzhaky, and Sharon Shoham. Abstraction-based interaction model for  
1033 synthesis. In Isil Dillig and Jens Palsberg, editors, *Verification, Model Checking, and Abstract*  
1034 *Interpretation*, pages 382–405, Cham, 2018. Springer International Publishing.

- 1036 40 Hila Peleg, Sharon Shoham, and Eran Yahav. Programming not only by example. In  
1037 *Proceedings of the 40th International Conference on Software Engineering*, pages 1114–1124.  
1038 ACM, 2018.
- 1039 41 Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati.  
1040 Scaling up superoptimization. In *ACM SIGARCH Computer Architecture News*, volume 44,  
1041 pages 297–310. ACM, 2016.
- 1042 42 Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program  
1043 synthesis. *ACM SIGPLAN Notices*, 50(10):107–126, 2015.
- 1044 43 Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language  
1045 models. In *ACM SIGPLAN Notices*, volume 49, pages 419–428. ACM, 2014.
- 1046 44 Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli.  
1047 cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In *Computer Aided*  
1048 *Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18,*  
1049 *2019, Proceedings, Part II*, pages 74–83, 2019.
- 1050 45 Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett.  
1051 Counterexample-guided quantifier instantiation for synthesis in smt. In *International Confer-*  
1052 *ence on Computer Aided Verification*, pages 198–216. Springer, 2015.
- 1053 46 Kensen Shi, Jacob Steinhardt, and Percy Liang. Frangel: Component-based synthesis with  
1054 control structures. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/  
1055 3290386.
- 1056 47 Rishabh Singh and Sumit Gulwani. Predicting a correct program in programming by  
1057 example. In *Computer Aided Verification - 27th International Conference, CAV 2015,*  
1058 *San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 398–414, 2015.  
1059 doi:10.1007/978-3-319-21690-4\_23.
- 1060 48 Rohit Singh, Venkata Vamsikrishna Meduri, Ahmed K. Elmagarmid, Samuel Madden, Paolo  
1061 Papotti, Jorge-Arnulfo Quiané-Ruiz, Armando Solar-Lezama, and Nan Tang. Synthesizing  
1062 entity matching rules by examples. *PVLDB*, 11(2):189–202, 2017. URL: <http://www.vldb.org/pvldb/vol11/p189-singh.pdf>.
- 1064 49 Calvin Smith and Aws Albarghouthi. Program synthesis with equivalence reduction. In  
1065 *Verification, Model Checking, and Abstract Interpretation - 20th International Conference,*  
1066 *VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings*, pages 24–47, 2019. doi:  
1067 10.1007/978-3-030-11245-5\_2.
- 1068 50 Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013. URL: <http://dx.doi.org/10.1007/s10009-012-0249-7>, doi:10.1007/s10009-012-0249-7.
- 1070 51 Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent  
1071 data structures. In *ACM SIGPLAN Notices*, volume 43, pages 136–148. ACM, 2008.
- 1072 52 Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat.  
1073 Combinatorial sketching for finite programs. *ACM SIGOPS Operating Systems Review*,  
1074 40(5):404–415, 2006.
- 1075 53 Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T. Freeman, Joshua B.  
1076 Tenenbaum, and Jiajun Wu. Learning to infer and execute 3d shape programs. In *7th*  
1077 *International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA,*  
1078 *May 6-9, 2019*, 2019.
- 1079 54 Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided  
1080 host languages. In *ACM SIGPLAN Conference on Programming Language Design and*  
1081 *Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 54, 2014.  
1082 URL: <http://doi.acm.org/10.1145/2594291.2594340>, doi:10.1145/2594291.2594340.
- 1083 55 Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin,  
1084 and Rajeev Alur. Transit: specifying protocols with concolic snippets. *ACM SIGPLAN Notices*,  
1085 48(6):287–296, 2013.

- 1086 **56** Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql  
1087 queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference*  
1088 *on Programming Language Design and Implementation*, pages 452–466. ACM, 2017.
- 1089 **57** Ke Wang, Rishabh Singh, and Zhendong Su. Search, align, and repair: data-driven feedback  
1090 generation for introductory programming exercises. In *ACM SIGPLAN Notices*, volume 53,  
1091 pages 481–495. ACM, 2018.
- 1092 **58** Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. Automated migration of hierarchical data  
1093 to relational tables using programming-by-example. *Proc. VLDB Endow.*, 11(5):580–593,  
1094 January 2018. doi:10.1145/3187009.3177735.
- 1095 **59** Tianyi Zhang, Di Yang, Crista Lopes, and Miryung Kirnt. Analyzing and supporting adaptation  
1096 of online code examples. In *Proceedings of the 41st International Conference on Software*  
1097 *Engineering*, pages 316–327. IEEE Press, 2019.