Augmented Example-Based Synthesis using Relational Perturbation Properties

SHENGWEI AN, Purdue University, USA
RISHABH SINGH, Google Brain, USA
SASA MISAILOVIC, UIUC, USA
ROOPSHA SAMANTA, Purdue University, USA

Example-based specifications for program synthesis are inherently ambiguous and may cause synthesizers to generate programs that do not exhibit intended behavior on unseen inputs. Existing synthesis techniques attempt to address this problem by either placing a domain-specific syntactic bias on the hypothesis space or heavily relying on user feedback to help resolve ambiguity.

We present a new framework to address the ambiguity/generalizability problem in example-based synthesis. The key feature of our framework is that it places a semantic bias on the hypothesis space using *relational perturbation properties* that relate the perturbation/change in a program output to the perturbation/change in a program input. An example of such a property is *permutation invariance*: the program output does not change when the elements of the program input (array) are permuted. The framework is portable across multiple domains and synthesizers and is based on two core steps: (1) automatically augment the set of user-provided examples by *applying* relational perturbation properties and (2) use a generic example-based synthesizer to generate a program consistent with the augmented set of examples. Our framework can be instantiated with three different user interfaces, with varying degrees of user engagement to help infer relevant relational perturbation properties. This includes an interface in which the user only provides examples and our framework *automatically infers* relevant properties. We implement our framework in a tool SketchAX specialized to the Sketch synthesizer and demonstrate that SketchAX is effective in significantly boosting the performance of Sketch for all three user interfaces.

CCS Concepts: • **Software and its engineering → General programming languages**; *Correctness*.

Additional Key Words and Phrases: Program Synthesis, Example-Based Synthesis, Ambiguity-Resolution

ACM Reference Format:
Shengwei An, Rishabh Singh, Sasa Misailovic, and Roopsha Samanta. 2020. Augmented Example-Based Synthesis using Relational Perturbation Properties. *Proc. ACM Program. Lang.* 4, POPL, Article 56 (January 2020), 24 pages. https://doi.org/10.1145/3371124

1 INTRODUCTION

Example-based synthesis, or Programming By Examples [Gulwani et al. 2012; Lieberman 2000] is an emerging paradigm of program synthesis that has been applied successfully across diverse domains [Feser et al. 2015; Gulwani et al. 2012; Leung et al. 2015; Singh and Gulwani 2012; Singh and Solar-Lezama 2011; Smith and Albarghouthi 2016]. The task in example-based synthesis is to generate a program from a hypothesis space (often defined as a domain-specific language or DSL)

Authors' addresses: Shengwei An, Purdue University, USA, an93@purdue.edu; Rishabh Singh, Google Brain, USA, rising@google.com; Sasa Misailovic, UIUC, USA, misailo@illinois.edu; Roopsha Samanta, Purdue University, USA, roopsha@purdue.edu.

2475-1421/2020/1-ART56
https://doi.org/10.1145/3371124

that satisfies a set of input-output (I/O) examples. The example-based specification mechanism can be a double-edged sword. Example-based specifications have made program synthesis more tractable as well as accessible to non-expert users who may not be able to write formal/complete specifications. However, example-based specifications also pose some of the biggest challenges in example-based synthesis: *ambiguity-resolution* [Gulwani 2016] and the related problem of *generalizability*. Since examples are inherently an ambiguous and/or incomplete form of specification, there can be a large number of programs that are consistent with a set of examples. Unsurprisingly, not all of these programs exhibit the (implicit) intended behavior on unseen inputs and, hence, may fail to *generalize* to unseen inputs.

There are two main classes of techniques that have been used to address the ambiguity/generalizability problem in example-based synthesis, with some caveats. (1) *Syntactic bias*-based techniques use highly structured DSLs [Alur et al. 2013; Solar-Lezama et al. 2006] or ranking functions [Singh and Gulwani 2015] to place a syntactic bias on the hypothesis space. These solutions are either inadequate by themselves or too domain-specific. (2) *User feedback loop*-based techniques employ a user to validate candidate programs or abstract representations of examples, or answer questions as in active learning [Drachsler-Cohen et al. 2017; Mayer et al. 2015; Peleg et al. 2018]. While some of these interaction models, e.g., Drachsler-Cohen et al. [2017], are based on principled approaches to address the generalizability problem in example-based synthesis, they place a heavy burden on the user that ultimately limits the scope of usability of example-based synthesis.

In this paper, we present a new approach for addressing the ambiguity/generalizability problem in example-based synthesis. Our framework is portable across multiple domains and synthesizers, can be instantiated with different user interfaces (UIs), and can be used in conjunction with existing techniques based on structured DSLs, ranking functions or user feedback loops. The key feature of our framework is that it places a *semantic bias* on the hypothesis space based on *relational perturbation properties*. In contrast to general relational properties that may express constraints relating multiple programs or multiple executions of a single program, relational perturbation properties relate the perturbation/change in a program output to the perturbation/change in a program input. An example of such a property is *permutation invariance*: the program output does not change when the elements of the program input (array) are permuted.

Relational perturbation properties enable us to design a simple and efficient solution that is similar, at least in spirit, to *data augmentation* used for improving the generalizability of machine learning models [Krizhevsky et al. 2012; Simard et al. 2003]. Our core approach is based on two steps:

(1) automatically generate an augmented set of examples by *applying* relational perturbation properties to the user-provided examples, and
(2) use an existing example-based synthesizer to generate a program consistent with the augmented set of examples.

Our solution strategy of enforcing relational properties using examples instead of formal specifications is inspired by two observations: (i) not all example-based synthesizers (e.g. [Gulwani et al. 2012]) accept specifications over all inputs and (ii) in cases where an example-based synthesizer accepts such specifications, there is typically a significant performance penalty in terms of synthesis time. We choose relational perturbation properties as they enable us to easily generate additional examples from any set of user-provided examples. For instance, given an I/O example (x, y) consisting of an input array $x = [1, 2, 3]$ and an output $y = 3$, it is easy to generate additional examples by *applying* permutation invariance: $([3, 2, 1], 3)$, $([2, 1, 3], 3)$, and so on. On the other hand, if we were to use a more general relational property, such as associativity for a program P with two inputs and one output $(\forall x, x', x''.\ P(P(x, x'), x'') = P(x, P(x', x'')))$, the user-provided examples would need to meet several requirements to enable generation of additional examples.

For associativity, one would need the user-provided example set to include the examples $((x, x'), y)$, $((y, x''), z)$ and $((x', x''), r)$ in order to generate the single additional example $((x, r), z)$.

So, where do the relational perturbation properties come from? Our framework provides three ways to answer this question using three UIs, with varying degrees of user enagement. In the *Property-Selection UI*, the user picks relevant relational perturbation properties in addition to providing examples. In the *Property-Validation UI*, the user provides examples and helps our framework learn relevant properties by validating/invalidating a small set of examples. Finally, in the *Property-Inference UI*, which is identical to the standard example-based synthesis setting, the user only provides examples and our framework automatically infers a relevant set of properties using a Partial Max-SMT [Cimatti et al. 2010]-based formulation.

To evaluate the efficacy of our technique, we instantiate our approach on top of the Sketch synthesizer [Solar-Lezama et al. 2006] and implement it in a tool SketchAX.[1] We chose Sketch as it is a general-purpose synthesizer that supports different forms of specifications including input-output examples, partial programs, and reference implementations. Moreover, unlike PBE systems such as FlashFill [Gulwani 2011] and Scythe [Wang et al. 2017], Sketch is not specialized to custom domain-specific langauges and ranking heuristics. Our extensive evaluation on a large class of benchmarks demonstrates that SketchAX significantly boosts Sketch's ability to synthesize correct programs for all three UIs. For instance, for benchmarks that satisfy some relational perturbation properties, SketchAX improves the success rate of Sketch by 61%, 60% and 59%, respectively, for the three UIs.

Contributions. Our paper makes the following key contributions:

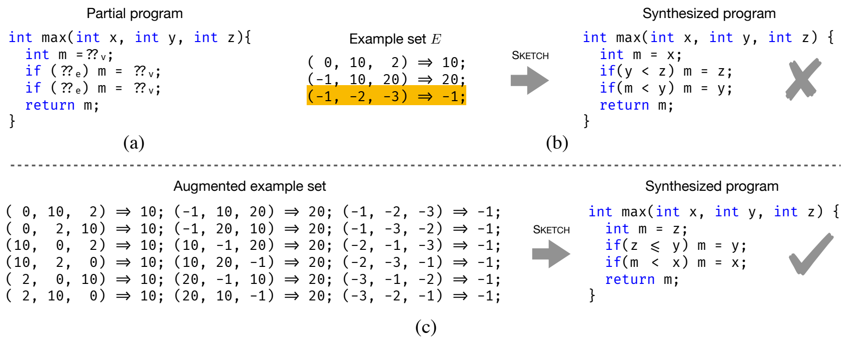
- We present a new approach to address the ambiguity/generalizability problem in example-based synthesis. Our approach is based on the novel idea of placing a semantic bias on the hypothesis space using relational perturbation properties (Sec. 4).
- We propose a flexible and portable framework that can be instantiated with three different UIs, with varying degrees of user engagement to help infer relevant properties (Sec. 5).
- We develop a Partial Max-SMT-based formulation to automatically infer relevant properties for the Property-Inference UI, where the user only provides I/O examples (Sec. 5).
- We implement our framework in a tool SketchAX specialized to the Sketch synthesizer (Sec. 6) and demonstrate that SketchAX is effective in significantly boosting the performance of Sketch for all three UIs (Sec. 7).

2 ILLUSTRATIVE EXAMPLES

We illustrate the core approach of our framework with a few motivating examples using the Sketch synthesizer.

max. Suppose a user wants to synthesize a `max` program that returns the maximum of 3 integer-valued inputs, using Sketch as an example-based synthesizer. A partial program (with holes) that the user may provide is shown in Fig. 1(a). The partial program encodes the space of expressions that can be used to fill each hole. For example, in the partial assignment statement `max = `$??_\mathsf{v}$, the construct $??_\mathsf{v}$ is shorthand for the *regular expression generator* `{|x|y|z|??|}` that defines a space of expressions equaling `x` or `y` or `z` or any integer constant. Similarly, the construct $??_\mathsf{e}$ is shorthand for the regular expression generator (|max|x|y|z|) (< | ≤) (max|x|y|z) that defines a space of Boolean expressions with either a < or ≤ operator, and operands equaling `max` or `x` or `y` or `z`. For the example set E in Fig. 1(b), despite the heavy syntactic bias placed on the hypothesis

[1] SketchAX is Sketch with Augmented Examples.



```
int max(int x, int y, int z){
  int m =??v;
  if (??e) m = ??v;
  if (??e) m = ??v;
  return m;
}
```

```
( 0, 10,  2) ⇒ 10;
(-1, 10, 20) ⇒ 20;
(-1, -2, -3) ⇒ -1;
```

```
int max(int x, int y, int z) {
  int m = x;
  if(y < z) m = z;
  if(m < y) m = y;
  return m;
}
```

```
( 0, 10,  2) ⇒ 10; (-1, 10, 20) ⇒ 20; (-1, -2, -3) ⇒ -1;
( 0,  2, 10) ⇒ 10; (-1, 20, 10) ⇒ 20; (-1, -3, -2) ⇒ -1;
(10,  0,  2) ⇒ 10; (10, -1, 20) ⇒ 20; (-2, -1, -3) ⇒ -1;
(10,  2,  0) ⇒ 10; (10, 20, -1) ⇒ 20; (-2, -3, -1) ⇒ -1;
( 2,  0, 10) ⇒ 10; (20, -1, 10) ⇒ 20; (-3, -1, -2) ⇒ -1;
( 2, 10,  0) ⇒ 10; (20, 10, -1) ⇒ 20; (-3, -2, -1) ⇒ -1;
```

```
int max(int x, int y, int z) {
  int m = z;
  if(z ⩽ y) m = y;
  if(m  <  x) m = x;
  return m;
}
```

Fig. 1. Computing the maximum of three integers using Sketch and SketchAX. The construct $??_\mathsf{v}$ is shorthand for the *regular expression generator* {|x|y|z|??|} that defines a space of expressions equaling x or y or z or any integer constant. The construct $??_\mathsf{e}$ is shorthand for the regular expression generator (|max|x|y|z|) (< | ≤) (max|x|y|z) that defines a space of Boolean expressions with either a < or ≤ operator, and operands equaling max or x or y or z.

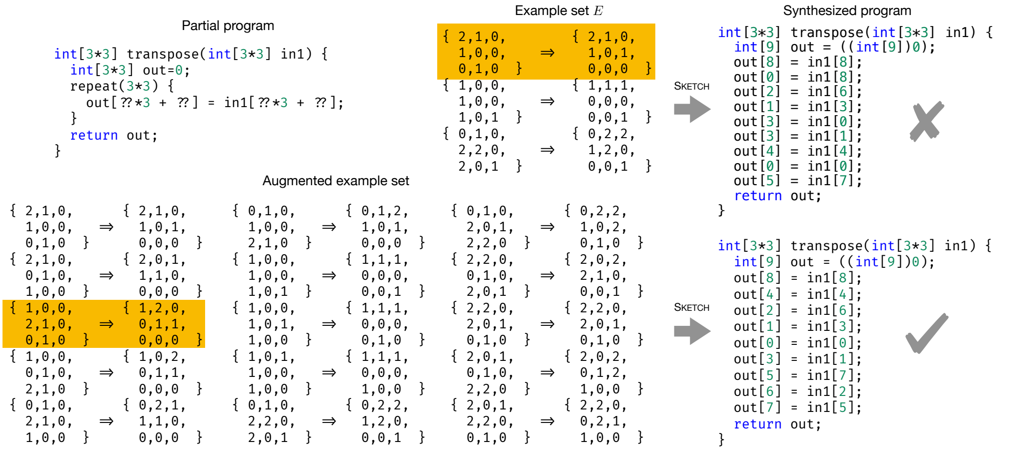
space by the partial program, Sketch fails to generate a correct max program. This illustrates the problem of ambiguity-resolution/generalizability in example-based synthesis, mentioned in Sec. 1. Our tool SketchAX addresses this problem by exploiting the fact that the max program should satisfy *permutation-invariance*: the program output should not change if we permute the program inputs. SketchAX automatically augments the initial set of examples E by *applying* the permutation-invariance property to the examples in E as shown in Fig. 1(c). With the additional semantic bias on the hypothesis space placed by this augmented set of examples, Sketch is able to generate the correct max program.

Permutation invariance is an instance of a *relational perturbation property* that relates perturbed inputs to corresponding perturbed outputs of programs. Specifically, it is a *structural* perturbation property which changes the relative *positions* of inputs and outputs. The max program also satisfies a *value* perturbation property (specifically, *value preservation*) which modifies the values of inputs and outputs. E.g. if we multiply all inputs by some positive constant integer, the output will also be multiplied by the same constant.

We formalize our notion of relational perturbation properties in Sec. 4. Next, we illustrate two useful structural and value perturbation properties.

matrixTranspose. The top half of Fig. 2 shows a partial program and an example set E used to synthesize a program to compute the transpose of a matrix. The program generated by Sketch is incorrect. From linear algebra, we know that if we permute the rows of the input matrix, the columns of its transpose will be permuted in the same way. SketchAX applies this perturbation property to E, thereby enabling Sketch to synthesize the correct program. For instance, the highlighted example in E in Fig. 2 is perturbed by swapping the top 2 rows of the input matrix and swapping the left 2 columns of the output matrix to yield the highlighted perturbed example.

arrayAdd. The top half of Fig. 3 shows a partial program and example set used to synthesize a program that performs the element-wise addition of two arrays in1 and in2. The program generated by Sketch is incorrect. SketchAX applies a value perturbation property to the examples, enabling Sketch to synthesize the correct program. Specifically, if in1 is perturbed by adding d_1 to each of its elements and in2 is perturbed by adding d_2 to each of its elements, each element of



```
int[3*3] transpose(int[3*3] in1) {
  int[3*3] out=0;
  repeat(3*3) {
    out[??*3 + ??] = in1[??*3 + ??];
  }
  return out;
}
```

```
{ 2,1,0,       { 2,1,0,
  1,0,0,   ⇒     1,0,1,
  0,1,0  }       0,0,0  }
{ 1,0,0,       { 1,1,1,
  1,0,0,   ⇒     0,0,0,
  1,0,1  }       0,0,1  }
{ 0,1,0,       { 0,2,2,
  2,2,0,   ⇒     1,2,0,
  2,0,1  }       0,0,1  }
```

```
int[3*3] transpose(int[3*3] in1) {
  int[9] out = ((int[9])0);
  out[8] = in1[8];
  out[0] = in1[8];
  out[2] = in1[6];
  out[1] = in1[3];
  out[3] = in1[0];
  out[3] = in1[1];
  out[4] = in1[4];
  out[0] = in1[0];
  out[5] = in1[7];
  return out;
}
```

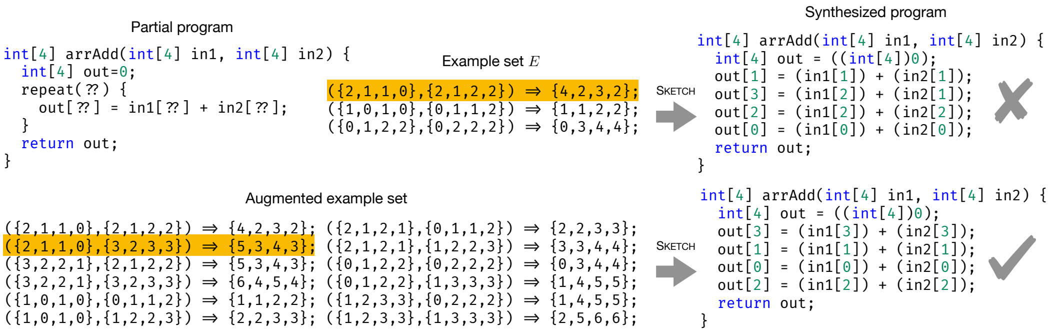
```
{ 2,1,0,       { 2,1,0,      { 0,1,0,       { 0,1,2,      { 0,1,0,       { 0,2,2,
  1,0,0,   ⇒     1,0,1,        1,0,0,   ⇒     1,0,1,        2,0,1,   ⇒     1,0,2,
  0,1,0  }       0,0,0  }      2,1,0  }       0,0,0  }      2,2,0  }       0,1,0  }
{ 2,1,0,       { 2,0,1,      { 1,0,0,       { 1,1,1,      { 2,2,0,       { 2,0,2,
  0,1,0,   ⇒     1,1,0,        1,0,0,   ⇒     0,0,0,        0,1,0,   ⇒     2,1,0,
  1,0,0  }       0,0,0  }      1,0,1  }       0,0,1  }      2,0,1  }       0,0,1  }
{ 1,0,0,       { 1,2,0,      { 1,0,0,       { 1,1,1,      { 2,2,0,       { 2,2,0,
  2,1,0,   ⇒     0,1,1,        1,0,1,   ⇒     0,0,0,        2,0,1,   ⇒     2,0,1,
  0,1,0  }       0,0,0  }      1,0,0  }       0,1,0  }      0,1,0  }       0,1,0  }
{ 1,0,0,       { 1,0,2,      { 1,0,1,       { 1,1,1,      { 2,0,1,       { 2,0,2,
  0,1,0,   ⇒     0,1,1,        1,0,0,   ⇒     0,0,0,        0,1,0,   ⇒     0,1,2,
  2,1,0  }       0,0,0  }      1,0,0  }       1,0,0  }      2,2,0  }       1,0,0  }
{ 0,1,0,       { 0,2,1,      { 0,1,0,       { 0,2,2,      { 2,0,1,       { 2,2,0,
  2,1,0,   ⇒     1,1,0,        2,2,0,   ⇒     1,2,0,        2,2,0,   ⇒     0,2,1,
  1,0,0  }       0,0,0  }      2,0,1  }       0,0,1  }      0,1,0  }       1,0,0  }
```

```
int[3*3] transpose(int[3*3] in1) {
  int[9] out = ((int[9])0);
  out[8] = in1[8];
  out[4] = in1[4];
  out[2] = in1[6];
  out[1] = in1[3];
  out[0] = in1[0];
  out[3] = in1[1];
  out[5] = in1[7];
  out[6] = in1[2];
  out[7] = in1[5];
  return out;
}
```

Fig. 2. Synthesizing a function to compute the transpose of a matrix using SKETCH and SKETCHAX.

the output array should be perturbed by $d_1 + d_2$. The perturbed examples shown in the bottom half of Fig. 3 are obtained using $d_1, d_2 \in \{0, 1\}$.

Remark. Here, we do not discuss the source of relational perturbation properties. Recall that our framework supports three UIs to help learn relevant properties. Our procedures for all UIs are presented in Sec. 5 and Sec. 6.



```
int[4] arrAdd(int[4] in1, int[4] in2) {
  int[4] out=0;
  repeat(??) {
    out[??] = in1[??] + in2[??];
  }
  return out;
}
```

```
({2,1,1,0},{2,1,2,2}) ⇒ {4,2,3,2};
({1,0,1,0},{0,1,1,2}) ⇒ {1,1,2,2};
({0,1,2,2},{0,2,2,2}) ⇒ {0,3,4,4};
```

```
int[4] arrAdd(int[4] in1, int[4] in2) {
  int[4] out = ((int[4])0);
  out[1] = (in1[1]) + (in2[1]);
  out[3] = (in1[2]) + (in2[1]);
  out[2] = (in1[2]) + (in2[2]);
  out[0] = (in1[0]) + (in2[0]);
  return out;
}
```

```
({2,1,1,0},{2,1,2,2}) ⇒ {4,2,3,2}; ({2,1,2,1},{0,1,1,2}) ⇒ {2,2,3,3};
({2,1,1,0},{3,2,3,3}) ⇒ {5,3,4,3}; ({2,1,2,1},{1,2,2,3}) ⇒ {3,3,4,4};
({3,2,2,1},{2,1,2,2}) ⇒ {5,3,4,3}; ({0,1,2,2},{0,2,2,2}) ⇒ {0,3,4,4};
({3,2,2,1},{3,2,3,3}) ⇒ {6,4,5,4}; ({0,1,2,2},{1,3,3,3}) ⇒ {1,4,5,5};
({1,0,1,0},{0,1,1,2}) ⇒ {1,1,2,2}; ({1,2,3,3},{0,2,2,2}) ⇒ {1,4,5,5};
({1,0,1,0},{1,2,2,3}) ⇒ {2,2,3,3}; ({1,2,3,3},{1,3,3,3}) ⇒ {2,5,6,6};
```

```
int[4] arrAdd(int[4] in1, int[4] in2) {
  int[4] out = ((int[4])0);
  out[3] = (in1[3]) + (in2[3]);
  out[1] = (in1[1]) + (in2[1]);
  out[0] = (in1[0]) + (in2[0]);
  out[2] = (in1[2]) + (in2[2]);
  return out;
}
```

Fig. 3. Synthesizing a function to compute the sum of two arrays using SKETCH and SKETCHAX.

3 PRELIMINARIES

We first define our models of programs and example-based synthesizers.

Programs. The semantics $[\![P]\!]$ of a program P is a function $[\![P]\!] : D_{in} \mapsto D_{out}$ mapping variables over an input domain D_{in} to variables over an output domain D_{out}. For simplicity of presentation, we assume that D_{in}, D_{out} range over arrays of integers. Our implementation can handle a wider variety of variable domains including scalars, arrays and matrices[2] over Booleans and integers. We use D^n to denote a domain of integer arrays of size n. We say a program P is *consistent* with an

[2] Matrices are modeled as arrays in our implementation.

input/output (I/O) example (x, y) if $[\![P]\!](x) = y$.

Equivalent programs. Two programs P and P' are *equivalent*, denoted $P \equiv P'$, if P and P' share the same input domain D_{in}, and, $\forall x \in D_{in}$. $[\![P]\!](x) = [\![P']\!](x)$.

Synthesizers. An example-based synthesizer, also sometimes referred to as a *synthesizer*, accepts as input a set E of I/O examples and generates a program P that is consistent with all examples in E. We sometimes refer to I/O examples simply as *examples.* Given a synthesizer S and a set E of examples, we use $S(E)$ to denote the program generated by S[3]. We assume that all user-provided examples are free of error and noise, i.e., all examples are consistent with the implicit specification a user may have in mind.

Some synthesizers are *constraint-based* – they accept constraints in first-order logic (modulo background theories) and use satisfiability modulo theory (SMT) solvers to generate a program that satisfies all constraints. Note that I/O examples can easily be encoded as constraints. Given a set C of constraints, we use $S(C)$ to denote the program generated by a constraint-based synthesizer S.

We say a constraint-based synthesizer S can solve a Partial Max-SMT problem if it can accept a set C^{hard} of constraints that are declared to be *hard* (i.e., non-relaxable) and a set C^{soft} of constraints declared to be *soft* (i.e., relaxable) and generate a program that satisfies all the hard constraints and maximizes the number of satisfied soft constraints. When such a synthesizer is used in its Max-SMT mode, we denote the synthesized program as $S(C^{hard}, C^{soft})$.

In what follows, unless explicitly stated, we *do not* assume a synthesizer to be constraint-based or to be able to solve a Partial Max-SMT problem.

4 RELATIONAL PERTURBATION PROPERTIES

We now formalize our notion of relational perturbation. We first present a fairly general *parametric* notion of relational perturbation and then present interesting instantiations that are used in our evaluation in Sec. 7.

Perturbation arrays and functions. We consider two classes of perturbation that can be applied to (integer) arrays: structural and value perturbation. A *structural perturbation function* applied to an array changes the positions of the array elements according to a given *structural perturbation array* of indices. A *value perturbation function* applied to an array changes the values of all array elements according to a given *value perturbation array* of parameters. Thus, a structural perturbation function does not modify the values of an array, a value perturbation function does not modify the positions of array elements, and neither perturbation function modifies the size of an array.

Definition 4.1 (Structural perturbation array). A structural perturbation array of size n, Q_n, is an array of indices in D^n: (1) $\forall i \in \{0, \ldots, n-1\}$. $Q_n[i] \in \{0, \ldots, n-1\}$ and (2) $\forall i, j \in \{0, \ldots, n-1\}$. $Q_n[i] = Q_n[j] \Rightarrow i = j$.

Definition 4.2 (Structural perturbation function). Let Q_n be a structural perturbation array. A Q_n-structural perturbation function $f^s_{Q_n} : D^n \mapsto D^n$ applied to an array $x \in D^n$ returns an array $x' \in D^n$ such that $\forall i \in \{0, \ldots, n-1\}$. $x[i] = x'[Q_n[i]]$.

Thus, a structural perturbation array is a permutation of the indices $\{0, \ldots, n-1\}$. and a structural perturbation function permutes the array elements according to a given structural perturbation array.

[3] We assume that a synthesizer is deterministic. While many synthesizers may execute nondeterministically, they often have options to force deterministic behavior. For instance, one can use the '--slv-seed' option for the synthesizer Sketch, use options '--seed' and '--random-seed' for the synthesizer CVC4, and run the synthesizer DryadSynth in a single-threaded mode to force determinism.

Example 4.3. In Fig. 1, the input array $[-1, -3, -2]$ in the highlighted example of E' can be obtained by applying the $[0, 2, 1]$-structural perturbation function to the input array $[-1, -2, -3]$ in the highlighted example of E. We write this as: $f^s_{[0,2,1]}([-1, -2, -3]) = [-1, -3, -2]$.

Example 4.4. The application of $f^s_{[n-1,n-2...,0]}$ to an array $x \in D^n$ returns an array that reverses the elements of x.

The set of all structural perturbation arrays of size n is denoted Q_n. Note that if $n = 1$, then $Q_1 = [0]$ and $f^s_{Q_1}(x) = x$ for any $x \in D^1$. We refer to the *identical* structural perturbation array $[0, 1, \ldots, n-1]$ as id^s_n. Thus, $f^s_{id^s_n}(x) = x$ for any $x \in D^n$. We refer to the structural perturbation array $[k+1, \ldots, n-1, 0, 1, \ldots, k]$, corresponding to a rotation to the right by k positions, as rot^k_n, and the complementary structural perturbation array, corresponding to a rotation to the left by k positions, as rot^{-k}_n.

Definition 4.5 (Value perturbation array). A value perturbation array, $V = [d_1, d_2]$, is an array of rational-valued parameters $d_1, d_2 \in \mathbb{Q}$.

Definition 4.6 (Value perturbation function). Given a value perturbation array $V = [d_1, d_2]$, a V-value perturbation function $f^v_V : D^n \mapsto D^n$ applied to an array $x \in D^n$ returns an array $x' \in D^n$ such that $\forall i \in \{0, \ldots, n-1\}.\ x'[i] = d_1 x[i] + d_2$.

Thus, a value perturbation function applies a specific affine transformation to every element of an array using parameters defined by a value perturbation array.

Example 4.7. The application of $f^v_{[2,1]}$ to the array $x = [2, 3, 5, 7]$ yields the array $y = [5, 7, 11, 15]$ and the application of $f^v_{[1/2,-1/2]}$ to y yields x again.

Example 4.8. While the current formalization is limited to single input arrays, we use the example from Figure 3 to illustrate how the formalization extends naturally to multiple input arrays. In Figure 3, the input arrays $([2, 1, 1, 0], [3, 2, 3, 3])$ in the highlighted perturbed example can be obtained by applying a $([1, 0], [1, 1])$-value perturbation function to the input arrays $([2, 1, 1, 0], [2, 1, 2, 2])$ in the highlighted example of E; the first input array is left unchanged and the elements of the second input array are incremented by 1.

Thus, a value perturbation function applies the same affine transformation to all elements of an array. The (infinite) set of all value perturbation arrays is denoted $\mathcal{V}$. We refer to the *identical* value perturbation array $[1, 0]$ as id^v.

Relational perturbation properties. We define *relational perturbation properties* to relate perturbed inputs to corresponding perturbed outputs of programs. We use A and f to denote both structural and value perturbation arrays and functions, respectively. We refer to a perturbation array and perturbation function applied to the input (output) of a program as an *input (output) perturbation array* A_{in} (A_{out}) and an *input (output) perturbation function* $f_{A_{in}}$ ($f_{A_{out}}$), respectively.

Henceforth, we fix the sizes of input and output arrays to be n, m, respectively.

Definition 4.9 (Relational perturbation property). A relational perturbation property R is a tuple $(K_1, K_2, \oplus, \mathcal{A}_{in})$ of a matrix K_1 of rationals, an array K_2 of rationals, an operator $\oplus$ and a set $\mathcal{A}_{in}$ of input perturbation arrays such that: for each $A_{in} \in \mathcal{A}_{in}$, the corresponding output perturbation array $A_{out} = K_1 A_{in} \oplus K_2$[4]. The operator $\oplus \in \{+, +_m\}$ where $+$ is addition and $+_m$ is addition modulo m.

[4] For convenience, we assume arrays are column vectors.

The above definition of a relational perturbation property is very general and can potentially be instantiated in infinitely many ways using its parameters K_1, K_2 and $\mathcal{A}_{in}$. We present two classes of interesting instantiations below that our evaluation focuses on (Sec. 7). In what follows, $0_{m\times n}$ denotes the zero matrix of size $m \times n$ and $\mathcal{I}_n$ denotes the identity matrix of size n.
Structural relational perturbation properties. These are perturbation properties where both the input and output perturbation is structural. Thus, each A_{in} is a column vector of size n, A_{out} is a column vector of size m, the matrix K_1 is of size $m\times n$, the array K_2 is a column vector of size m, and $\oplus = +_m$.

1. *Permutation invariance.* Permutation invariance specifies that the program output does not change when the elements of the program input (array) are permuted. Formally, for all $Q_n \in \mathcal{Q}_n$, we have $[\![P]\!](x) = [\![P]\!](f^s_{Q_n}(x))$. Permutation invariance is the relational perturbation property $(0_{m\times n}, id^s_m, +_m, \mathcal{Q}_n)$. Note, $\forall Q_n \in \mathcal{Q}_n$. $A_{out} = 0_{m\times n}\ Q_n\ +_m\ id^s_m$, i.e., $A_{out} = id^s_m$, as desired.
2. *Permutation preservation.* For this property, we assume that the sizes of input and output arrays are the same, i.e., $n = m$. Permutation preservation specifies that when the elements of the program input are permuted, the elements of the program output are permuted in the same way. Formally, for all $Q_n \in \mathcal{Q}_n$, we have $f^s_{Q_n}([\![P]\!](x)) = [\![P]\!](f^s_{Q_n}(x))$. This can be represented as the relational perturbation property $(\mathcal{I}_n, 0_{n\times 1}, +_n, \mathcal{Q}_n)$.
3. $(k, -k)$*-rotation.* For this property, we also assume that $n = m$. $(k, -k)$-rotation specifies that when the elements of the program input are rotated to the right by k positions, the elements of the program output are rotated to the left by k positions. Formally, for all $k \in \{-(N-1), \ldots, N-1\}$, we have $f^s_{rot^{-k}_n}([\![P]\!](x)) = [\![P]\!](f^s_{rot^k_n}(x))$. This can be represented as the relational perturbation property $(\mathcal{I}_n, k_{n\times 1}, +_n, \{rot^k_n \mid k \in \{-(N-1), \ldots, N-1\}\})$, where $k_{n\times 1}$ is a column vector of size n with all elements equal to k.

Value relational perturbation properties. These are perturbation properties where both the input and output perturbation are value perturbations. Here, A_{in}, A_{out} and the array K_2 are all column vectors of size 2, the matrix K_1 is of size 2×2, and $\oplus = +$.

1. *Value invariance.* This value relational perturbation property is similar to permutation invariance with the structural input perturbation replaced by a value input perturbation. Formally, Value invariance specifies that for all $V \in \mathcal{V}$, we have $[\![P]\!](x) = [\![P]\!](f^v_V(x))$. Value invariance can be represented by the relational perturbation property $(0_{2\times 2}, id^s_2, +, \mathcal{V})$. $A_{out} = id^v$ for all $V \in \mathcal{V}$ as desired.
2. *Value preservation.* This value relational perturbation property is similar to permutation preservation with the structural input perturbation replaced by a value input perturbation. Formally, Value preservation specifies that for all $V \in \mathcal{V}$, we have $f^v_V([\![P]\!](x)) = [\![P]\!](f^v_V(x))$ and can be represented as the relational perturbation property $(\mathcal{I}_2, 0_{2\times 1}, +, \mathcal{V})$. have $A_{out} = V$ for all $V \in \mathcal{V}$ as desired.

We define two additional value perturbation properties that are used in our evaluation (Sec. 7): $\mathcal{V}_{given}$-value invariance and $\mathcal{V}_{given}$-value preservation. These restrict the focus to a given set $\mathcal{V}_{given}$ of value perturbation arrays, instead of the set $\mathcal{V}$ of all possible value perturbation arrays.

Relational perturbation functions. Relational perturbation functions capture the notion of *applying* a relational perturbation property R to an example set E. Informally, the application of an R-relational perturbation function to E yields a *perturbed* example set E_{pert} obtained by perturbing each example in E according to R.

Definition 4.10 (Relational perturbation function). Given relational perturbation property $R = (K_1, K_2, \oplus, \mathcal{A}_{in})$, an R-relational perturbation function $f_R : (D^n, D^m) \mapsto (D^n, D^m)$ applied to an example set E returns an example set E_{pert} such that $(x', y') \in E_{pert}$ iff there exist $(x, y) \in E$ and $A_{in} \in \mathcal{A}_{in}$ such that $x' = f_{A_{in}}(x)$ and $y' = f_{A_{out}}(y)$ with $A_{out} = K_1 A_{in} \oplus K_2$.

5 ALGORITHMIC FRAMEWORK

We now present our overall solution framework to improve the generalizability of existing example-based synthesizers. Our framework supports three different UIs that differ in the degree of user involvement in identifying suitable relational perturbation properties for an example-based synthesis problem. The solutions for the first two UIs, i.e., the Property-Selection and Property-Validation UIs, are applicable to any example-based synthesizer that can handle the variable domains described in Sec. 3 and Sec. 4. The solution presented in this section for the third UI, i.e., the Property-Inference UI, is applicable to constraint-based synthesizers that can solve the Partial Max-SMT problem and can handle the variable domains described in Sec. 3 and Sec. 4. In Sec. 6, we describe how to specialize these solutions to the Sketch synthesizer and additionally present an alternate solution for the Property-Inference UI that does not require the synthesizer to be constraint-based (or be able to solve the Partial Max-SMT problem).

Algorithm 1: Example Augmentation

```
procedure PerturbExamples($E$, $R$)
    Input  : $E$: a set of I/O examples
             $R = (k_1, k_2, \oplus, \mathcal{A}_{in})$: a relational perturbation property
    Output : $E_{pert}$: a set of I/O examples obtained by applying $R$ to $E$
    $E_{pert} = \emptyset$
    foreach $(x, y) \in E$ do
        foreach $A_{in} \in \mathcal{A}_{in}$ do
            $E_{pert} = E_{pert} \cup \{(f_{A_{in}}(x),\ f_{k_1 A_{in} \oplus k_2}(y))\}$
    return $E_{pert}$
```

We begin with a procedure that implements the core strategy of our framework: *augment user-provided example sets by applying relational perturbation properties.* Given an example set E and a relational perturbation property R, this simple procedure, shown in Algo. 1, perturbs each example in E by applying R to it according to Def. 4.10. In what follows, we restrict our focus to a *finite* set of relational perturbation properties.

5.1 Augmented Synthesis: Property-Selection UI

In the Property-Selection UI, the user provides an example set E and a finite set $\mathcal{R}$ of relational perturbation properties. Our solution for this UI is shown in Algo. 2. We explicitly identify user inputs/interactions in a procedure by underlining them. Besides E and $\mathcal{R}$, the procedure also requires as input a synthesizer S. Unlike E and $\mathcal{R}$ which are user-provided inputs (hence, underlined), the synthesizer S is a tunable *parameter* of our framework.

Given these inputs, Algo. 2 generates a program consistent with examples in E. The procedure first uses Algo. 1 to obtain an augmented example set E_{aug} by perturbing the examples in E with all the properties in $\mathcal{R}$. Then, the procedure invokes synthesizer S using E_{aug} to generate the output program.

Algorithm 2: Augmented Synthesis: Property-Selection UI

```
procedure AugmentSynthesis_PropertySelection($\underline{E}, \underline{\mathcal{R}}, S$)
    Input  : $E$: a set of I/O examples
             $\mathcal{R}$: a set of relational perturbation properties
             $S$: an example-based synthesizer
    Output : $P$: a program consistent with examples in $E$
    $E_{aug} = E$
    for $R \in \mathcal{R}$ do
        $E_{pert}$ = PerturbExamples($E, R$)
        $E_{aug} = E_{aug} \cup E_{pert}$
    return $S(E_{aug})$
```

Algorithm 3: Augmented Synthesis: Property-Validation UI

```
procedure AugmentSynthesis_PropertyValidation($\underline{E}, \mathcal{R}, S, n$)
    Input  : $E, \mathcal{R}, S$: as before
             $n$: the number of perturbed examples shown to a user
    Output : $P$: a program consistent with examples in $E$
    $E_{aug} = E$
    for $R \in \mathcal{R}$ do
        $E_{pert}$ = PerturbExamples($E, R$)
        $E_{rand}$ = RandomlyChoose($E_{pert}, n$)
        if $\underline{\texttt{UserAccept}(E_{rand})}$ then
            $E_{aug} = E_{aug} \cup E_{pert}$
    return $S(E_{aug})$
```

5.2 Augmented Synthesis: Property-Validation UI

In the Property-Validation UI, the user provides an example set E and interacts with our framework to validate/invalidate perturbed examples. The user burden in this case is less than in the Property-Selection UI — instead of picking applicable relational perturbation properties, the user only needs to examine examples. As before, the user inputs/interactions are underlined in our procedure, Algo. 3, for this UI. The procedure is additionally parameterized by a synthesizer S, a set of relational perturbation properties $\mathcal{R}$, and the number n of user interactions per property.

For each property in $\mathcal{R}$, Algo. 3 uses Algo. 1 to generate a set E_{pert} of perturbed examples. Then, a set of n randomly chosen perturbed examples from E_{pert} are shown to the user. If the user accepts all n perturbed examples, the example set E is augmented with the perturbed examples E_{pert}. The procedure invokes synthesizer S using the final augmented example set E_{aug} to generate the output program. Notice that, for each property, Algo. 3 only requires a user to accept n perturbed examples in order to augment E with the entire set E_{pert} of perturbed examples corresponding to that property.

5.3 Augmented Synthesis: Property-Inference UI

In the Property-Inference UI, the user only provides an example set E. The user burden in this case is obviously the least among all our UIs. In fact, there is no additional burden on the user beyond a standard example-based synthesis setting. Not surprisingly, this UI is the most challenging for our framework as we need to automatically *infer* relevant relational perturbation properties without

Algorithm 4: Augmented Synthesis: Property-Inference UI

```
1 procedure AugmentSynthesis_PropertyInference($\underline{E}, \mathcal{R}, S$)
      Input  : $E, \mathcal{R}$: as before
               $S$: a constraint-based synthesizer that can solve a Partial Max-SMT problem
      Output : $P$: a program consistent with examples in $E$
2     $C^{hard} = \{(P(x) = y) \mid (x, y) \in E\}$
3     $C^{soft} = \emptyset$
4     for $R \in \mathcal{R}$ do
5         $E_{pert}$ = PerturbExamples($E, R$)
6         $C_R = \bigwedge_{(x,y) \in E_{pert}} (P(x) = y)$
7     $C^{soft} = \{C_1, \ldots, C_{|\mathcal{R}|}\}$
8     $\mathcal{L}$ = AllMaxSMTSol($S, C^{hard}, C^{soft}$)
      /* $\mathcal{R}^{\mathcal{L}}$: the set of property sets in $\mathcal{L}$                */
9     return AugmentSynthesis_PropertySelection($E$, Rank($\mathcal{R}^{\mathcal{L}}$), $S$)
```

any help from a user. Our solution, based on a Partial Max-SMT formulation, is shown in Algo. 4. Besides the (underlined) user-provided example set, the procedure is parameterized by a synthesizer S and a set of relational perturbation properties $\mathcal{R}$. We require S to be a constraint-based synthesizer that can solve a Partial Max-SMT problem.

For each example in E, the procedure generates a corresponding hard constraint. For each property $R \in \mathcal{R}$, the procedure generates a soft constraint corresponding to the set of perturbed examples obtaining by applying R to E (using Algo. 1). Once all constraints are generated, we have a Partial Max-SMT synthesis problem defined by the tuple (S, C^{hard}, C^{soft}). A solution $(P, \mathcal{R}^*)$ to this partial Max-SMT synthesis problem consists of a program P, synthesized by S, which is consistent with the set of all examples in E (i.e., C^{hard}) and all examples perturbed according to some maximal subset of properties $\bar{\mathcal{R}} \subseteq 2^{\mathcal{R}}$ (corresponding to a maximally satisfiable set of soft constraints in C^{soft}). In general, there can be multiple such solutions, say $\{(P_1, \bar{\mathcal{R}}_1), \ldots, (P_t, \bar{\mathcal{R}}_t)\}$; let us denote this set by $\mathcal{L}$. If Algo. 4 were to simply return $S(C^{hard}, C^{soft})$, this would be a program P corresponding to an arbitrary solution $(P, \bar{\mathcal{R}})$ from $\mathcal{L}$ (based on the search strategy of S). In particular, P may not be the most generalizable program and $\bar{\mathcal{R}}$ may not be the most suitable property set for the given example-based synthesis problem. While it is not clear how to formally define optimality of solutions to the Partial Max-SMT synthesis problem, Algo. 4 uses a more sophisticated approach than simply returning $S(C^{hard}, C^{soft})$.

First, Algo. 4 uses a procedure AllMaxSMTSol to obtain the entire set $\mathcal{L}$ of Partial Max-SMT synthesis solutions (Line 8). Let $\mathcal{R}^{\mathcal{L}}$ denote the set of property sets in $\mathcal{L}$. In Line 9, Algo. 4 uses a procedure Rank to obtain the property set in $\mathcal{R}^{\mathcal{L}}$ ranked highest by a ranking function and invokes Algo. 2 with this highest ranked property set.

The procedures AllMaxSMTSol and Rank can be instantiated in many ways. We describe our specific implementations in Sec. 6.

5.4 Correctness

An example-based synthesizer S is *sound* with respect to a set of examples E if: whenever S generates a program P from the set E of examples, P is guaranteed to be consistent with all examples in E. An example-based synthesizer S is *complete* with respect to E if: whenever there exists a program in S's hypothesis space consistent with the examples in E, S can always generate such a program.

```
int[N] F(int[N] in){
  ?? // holes in unknown program
  return out;
}
harness void userProvidedExamples(E){
  for (in,out) ∈ E:
      assert out == F(in);
}
harness void augmentedExamples(E_pert){
  for (in,out) ∈ E_pert:
      assert out == F(in);
}
```

Fig. 4. Sketch encoding for Algo. 1.

The example augmentation does not affect the soundness/completeness with respect to the user-provided set of examples:

Theorem 5.1. *The synthesis procedures in Algo. 2, Algo. 3 and Algo. 4 are sound and complete with respect to the user-provided example set E if the synthesizer S is sound and complete with respect to E.*

The theorem follows directly from the fact that the augmented example sets used by all three synthesis procedures include the user-provided examples and from our assumption that a user does not make mistakes: all user-provided and validated examples are free of error.

While we do not provide formal guarantees about the generalizability of the programs synthesized by our procedures, as we will see in Sec. 7, all our procedures can significantly improve the generalizabity of the Sketch synthesizer.

6 SKETCHAX

In this section, we describe the key components of the specialization, SketchAX, of our framework to the Sketch synthesizer. We present the basic Sketch encoding of our algorithms and an efficient alternative to `AugmentSynthesis_PropertyInference` that can also be used with example-based synthesizers that are not constraint-based.

Basic Sketch encoding. The main idea of the Sketch encoding for all our algorithms (see Fig. 4 for the Sketch encoding for Algo. 1) is to use the `harness` function in Sketch to impose I/O constraints, corresponding to user-provided and augmented examples, as `assert` statements.

Implementation of `AugmentSynthesis_PropertyInference`. When using an exact encoding of Algo. 4 in Sketch, we found that Sketch often struggles to complete the difficult Partial Max-SMT optimization problem within a specified time bound (even for returning one solution). Hence, we encode a simple greedy procedure (see Algo. 5) for solving the maximization constraint: instead of considering all properties in $\mathcal{R}$ at once, the procedure performs a greedy search over subsets of properties in $\mathcal{R}$ of increasing sizes. The procedure marks a property set as *satisfiable* if `AugmentSynthesis_PropertySelection` can successfully synthesize a program using the property set and Sketch, within a time bound. The procedure maintains such largest satisfiable subsets of properties until no subset can be expanded any further. The procedure generates all solutions

Algorithm 5: Greedy Implementation for Algo. 4

```
1  procedure GreedyAugmentSynthesis_PropertyInference($\underline{E}, \mathcal{R}, S, T$)
       Input  : $E, \mathcal{R}$: as before
                $S$: an example-based synthesizer
                $T$: time bound
       Output : $P$: a program synthesized with highest ranked properties
2      $\mathcal{R}_{app} = \emptyset$
3      for $R \in \mathcal{R}$ do
4          if AugmentSynthesis_PropertySelection_T($E, \{R\}, S$) $\neq$ None then
5              $\mathcal{R}_{app} = \mathcal{R}_{app} \cup \{R\}$
6      if $|\mathcal{R}_{app}| <= 1$ then
7          return AugmentSynthesis_PropertySelection($E, \mathcal{R}_{app}, S$)
8      $\widetilde{\mathcal{R}}_{sat} = \{\{R\} \mid R \in \mathcal{R}_{app}\}$
9      $conflict = \emptyset$
10     while $|\widetilde{\mathcal{R}}_{sat}| > 1$ do
11         $\widetilde{\mathcal{R}}_{newsat} = \emptyset$
12         for $\mathcal{R}_{oldsat} \in \widetilde{\mathcal{R}}_{sat}$ do
13             for $R_{new} \in \{R \mid R \in \mathcal{R}_{app} \wedge R \notin \mathcal{R}_{oldsat}\}$ do
14                 if $\exists R \in \mathcal{R}_{oldsat} \,.\, (R, R_{new}) \in conflict$ then
15                     continue
16                 if AugmentSynthesis_PropertySelection_T($E, \mathcal{R}_{oldsat} \cup \{R_{new}\}, S$) $\neq$ None
                   then
17                     $\widetilde{\mathcal{R}}_{newsat} = \widetilde{\mathcal{R}}_{newsat} \cup \{\mathcal{R}_{oldsat} \cup \{R_{new}\}\}$
18                 else
19                     if $|\mathcal{R}_{oldsat}| = 1$ then
20                         $conflict = conflict \cup \{(R_{new}, R), (R, R_{new}) \mid R \in \mathcal{R}_{oldsat}\}$
21         if $|\widetilde{\mathcal{R}}_{newsat}| = 0$ then
22             break
23         $\widetilde{\mathcal{R}}_{sat} = \widetilde{\mathcal{R}}_{newsat}$
24     return AugmentSynthesis_PropertySelection($E$, Rank($\widetilde{\mathcal{R}}_{sat}$), $S$)
```

to the Partial Max-SMT problem that can each be computed as being satisfiable within the time bound and does not require the synthesizer to even be constraint-based.[5]

In order to take a closer look at Algo. 5, we first define some notation used in the algorithm. Symbols R and R_{new} denote a single property, symbols $\mathcal{R}_{app}$ and $\mathcal{R}_{oldsat}$ denote a set of properties, and symbols $\widetilde{\mathcal{R}}_{sat}$ and $\widetilde{\mathcal{R}}_{newsat}$ denote a set of sets of properties. Further, given a property R, $\{R\}$ denotes the singleton set containing R. Finally, `AugmentSynthesis_PropertySelection_T` denotes a version of `AugmentSynthesis_PropertySelection` that is forced to return "None" after a time bound T is exceeded.

We first collect all individual, satisfiable/applicable properties in $\mathcal{R}_{app}$ (Lines 2–5). We do this by using `AugmentSynthesis_PropertySelection_T` to synthesize using each possible property

[5] We use a Partial Max-SMT formulation for Algo. 4, instead of the above greedy formulation, for ease of presentation. The choice between Algo. 4 and the greedy implementation is a practical one, and can be made based on the availability/efficiency of Partial Max-SMT-solving within the synthesizer in question.

$R \in \mathcal{R}$. If no property is found to be satisfiable or only one property is found to be satisfiable, Algo. 5 simply returns the program synthesized by `AugmentSynthesis_PropertySelection` with $\mathcal{R}_{app}$ (Lines 6–7). Then, we start to build maximal sets of satisfiable properties by adding one property from $\mathcal{R}_{app}$ iteratively to current satisfiable property sets (Lines 8—23). The set of maximal, satisfiable property sets, $\widetilde{\mathcal{R}}_{sat}$, is initialized with singleton sets containing each satisfiable property in $\mathcal{R}_{app}$ (Line 8). While there are at least two satisfiable property sets in $\widetilde{\mathcal{R}}_{sat}$ (Line 10), we iteratively try to construct a larger satisfiable property set $\widetilde{\mathcal{R}}_{newsat}$ with one additional property (Line 12—20). Thus, in each iteration, we choose a satisfiable property set $\mathcal{R}_{oldsat}$ and try to add a new property R_{new} into it. To be more efficient, we maintain a (symmetric) conflict relation between each pair of properties that essentially tracks if `AugmentSynthesis_PropertySelection_T` is able to synthesize a program using the pair of properties, within the time bound. If any property in $\mathcal{R}_{oldsat}$ conflicts with R_{new}, we just skip trying R_{new} (Lines 14—15). If the newly constructed property set $\mathcal{R}_{oldsat} \cup \{R_{new}\}$ is found to be satisfiable, we add it to the set of maximal, satisfiable property sets constructed in the current iteration of the outer while loop (Lines 16—17). Otherwise, we update the conflict relation (Lines 19—20). If we fail to construct a larger satisfiable property set, we exit the while loop with the previous set of maximal, satisfiable property sets in $\widetilde{\mathcal{R}}_{sat}$ (Lines 21—22). Otherwise, we start the next iteration of the while loop by updating $\widetilde{\mathcal{R}}_{sat}$ to the newly constructed set of maximal, satisfiable property sets $\widetilde{\mathcal{R}}_{newsat}$ (Line 23).

After collecting the maximal, satisfiable property sets, Algo. 5 returns the program synthesized by `AugmentSynthesis_PropertySelection` using the property set ranked highest by a ranking function (Line 24). In our actual implementation, we use dynamic programming to avoid using `AugmentSynthesis_PropertySelection` to synthesize programs with the same sets of examples and properties repeatedly in Line 24.

Ranking function. After analyzing the satisfiable property sets generated by the above greedy implementation of `AugmentSynthesis_PropertyInference` over our dataset and experimenting with different ranking functions (including one learnt from a training set of successful synthesis instances from the Property-Selection UI), we found that randomly choosing a property set from the set of satisfiable property sets is adequate for our experimental setup.

7 EVALUATION

Our evaluation of SketchAX investigates the improvement over Sketch in synthesis of correct benchmarks and the run-time performance of our algorithms for all three UIs. In what follows, the implementations in SketchAX of the algorithms for the three UIs are denoted SketchAX I, SketchAX II and SketchAX III, respectively.

Dataset. Our dataset consists of 143 "Sketch benchmarks" from Sketch Benchmarks Repositories [ske 2019], 40 "SyGuS benchmarks" from the Conditional Linear Integer Arithmetic (CLIA) track of the annual Syntax-Guided Synthesis (SyGuS) competition [syg 2019; Alur et al. 2013], and 14 manually-constructed benchmarks. Overall, our dataset consists of 111 *Bit* benchmarks which only use Booleans/bit-vectors and 86 *Int* benchmarks which only use integers/integer arrays.

The Sketch benchmarks were automatically selected from the set of all benchmarks available in [ske 2019] using a script based on the following requirements: (1) there is a reference implementation or complete functional specification for the benchmark; (2) the benchmark contains at least one hole; (3) Sketch can synthesize a program from the benchmark; (4) and the input/output of the benchmark are of types Boolean/integer or their arrays. This yielded 111 Bit benchmarks and 32 Int benchmarks. Note that we require reference implementation for benchmarks only for evaluating the correctness of the synthesized program in our experiments.

For the SyGuS benchmarks, we considered the CLIA track and the Programming By Examples [Theory of Bit Vectors] (PBE-BV) track as these tracks target synthesis of programs over Booleans/integers and their arrays. Unfortunately, we were unable to use the benchmarks from the PBE-BV track as they did not come with complete functional specifications or reference implementations that would enable us to check the correctness of the synthesized programs. We hand-translated all benchmarks in the CLIA track into Sketch and selected the ones for which Sketch was able to synthesize a program. This yielded 40 Int benchmarks.

Finally, we manually constructed an additional 14 Int benchmarks in Sketch to further enlarge our set of Int benchmarks. These include the three examples from Sec. 2, some variations on the partial programs in the Sketch benchmarks, and common algorithmic tasks over arrays such as computing the sum of two matrices, counting the number of elements in an array within/exceeding a threshold, computing the maximum element in an arbitrary-length input array, etc.

We focus on the following set of relational perturbation properties as they suffice for the benchmarks we considered:

- p_1: permutation invariance,
- p_2: permutation preservation,
- p_3: $(k, -k)$-rotation,
- p_4: $\mathcal{V}_{add}$-value invariance,
- p_5: $\mathcal{V}_{mult}$-value invariance,
- p_6: $\mathcal{V}_{add}$-value preservation,
- p_7: $\mathcal{V}_{mult}$-value preservation, and
- p_8: permutation transposition.

Here, $\mathcal{V}_{add}$ is $\{[1, d] \mid d \in \{1, \ldots, 10\}\}$ and $\mathcal{V}_{mult}$ is $\{[d, 0] \mid d \in \{2, \ldots, 10\}\}$. Thus, property p_4 perturbs inputs by adding $d \in \{1, \ldots, 10\}$ to all elements and leaves the output unchanged, property p_5 perturbs inputs by multiplying all elements with $d \in \{2, \ldots, 10\}$ and leaves the output unchanged, property p_6 perturbs inputs/outputs by adding $d \in \{1, \ldots, 10\}$ to all elements and property p_7 perturbs inputs/outputs by multiplying all elements with $d \in \{2, \ldots, 10\}$. Property p_8, illustrated in Fig. 2, applies an identical permutation to the rows and columns of input and output matrices, respectively. Notice that the set of all relational properties is finite because of the finite sets $\mathcal{V}_{add}$ and $\mathcal{V}_{mult}$ of value perturbation arrays.

Table 1. Number of properties satisfied by benchmarks in dataset

Number of properties satisfied	Bit benchmarks	Int benchmarks
1	22	12
2	0	28
3	0	12
4	0	1
>4	0	0
≥ 1	22	53

In Table 1, we provide a property-based view of our dataset. In particular, we show the number of Bit and Int benchmarks satisfying some number of properties from $p_1, \ldots, p_8$. Note that the Bit benchmarks satisfy at most one (structural perturbation) property, p_1 or p_2 or p_3, at a time and do not satisfy any value perturbation properties.

Experimental setup. We use the reference implementations or complete functional specifications of the benchmarks to check the correctness of synthesized programs (using the keyword `implements` in Sketch for checking program equivalence). Note that Sketch does bounded verification, i.e., it checks program equivalence over all inputs upto certain bounds (e.g. all integers upto bit width 5, all arrays upto length 10 etc.).

In our default setting, we evaluate the success rate of Sketch and our algorithms over 10 runs for each benchmark, yielding *1970 synthesis instances* in total. Each run uses a different set of 3 I/O examples, randomly generated from the complete functional specification/reference implementation. We set a timeout of 5 minutes for synthesis-solving across all experiments. For the perturbation of an I/O example with a relational perturbation property (Line 5 in `AugmentSynthesis_PropertySelection`), we set a timeout of 5 seconds and an upper-bound of 128 perturbed examples to ensure Sketch terminates within a reasonable time. On average, SketchAX I generated 112 examples for each benchmark satisfying some properties (157 for each int benchmark, and 93 for each bit benchmark).

We use the latest version of Sketch, released in March 2018 (Sketch-1.7.5). We ran our experiments on shared servers equipped with Intel E5320@1.86GHz CPU and 8GB RAM.

7.1 SketchAX I: Property-Selection UI

Figures 5 and 6 summarize the results of our synthesis experiments with Sketch and SketchAX I. For each benchmark, the applicable relational perturbation properties (from $p_1, \ldots, p_8$) are manually chosen by us. Fig. 5 shows the *instance success rate* for different benchmark categories. Recall that there are 10 synthesis instances per benchmark. The instance success rate is the percentage of synthesis instances that yield correct synthesized programs. The numbers below each benchmark category on the x-axis indicate the number of benchmarks in that category. Fig. 6 shows the *benchmark success rate* for different benchmark categories. A benchmark synthesis is declared successful if 100% of its synthesis instances yield correct synthesized programs. The benchmark success rate is then simply computed as the percentage of benchmarks whose synthesis is successful. We later investigate other thresholds for defining a successful benchmark synthesis (Fig. 7).

Let us first take a closer look at Fig. 5. The numbers within the SketchAX I bars indicate the improvement over the instance success rate of Sketch with SketchAX I. The overall improvement in the instance success rate of Sketch with SketchAX I is **22**%. The improvement is significantly higher (**61**%) for benchmarks which satisfy at least one of the relational perturbation properties $p_1, \ldots, p_8$, thereby validating our fundamental hypothesis. The improvement is even more considerable (**191**%) for Bit benchmarks satisfying some perturbation properties, a category in which Sketch's instance success rate is only 31%. The improvement for Int benchmarks satisfying some properties, a category for which Sketch's instance success rate is 58%, is **32**%. It is important to note that for benchmarks that do not satisfy any properties, SketchAX's success rate does not fall below that of Sketch.

The improvements in benchmark success rates, shown in Fig. 6, are slightly higher: **28**% for all benchmarks, **63**% for benchmarks satisfying some properties, **220**% for Bit benchmarks satisfying some properties, and **32**% for Int benchmarks satisfying some properties. Notice that, as expected, the individual benchmark success rates of Sketch and SketchAX I are lower than their respective instance success rates.

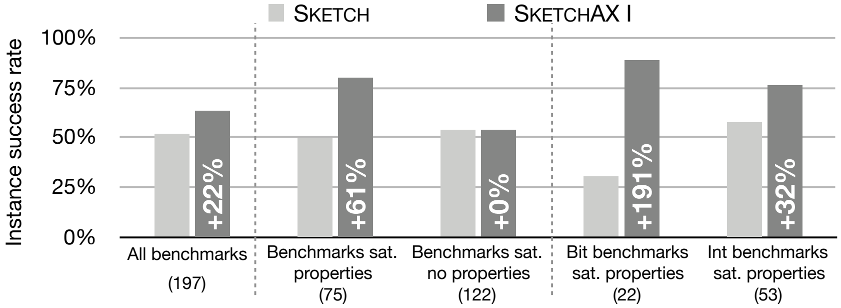


Fig. 5. Instance success rate of Sketch and SketchAX I.

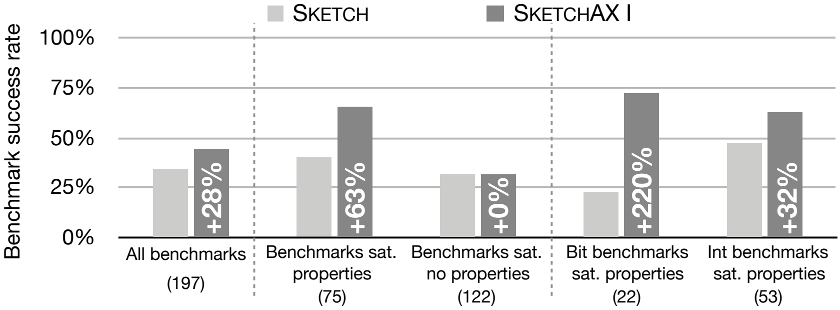


Fig. 6. Benchmark success rate of Sketch and SketchAX I.

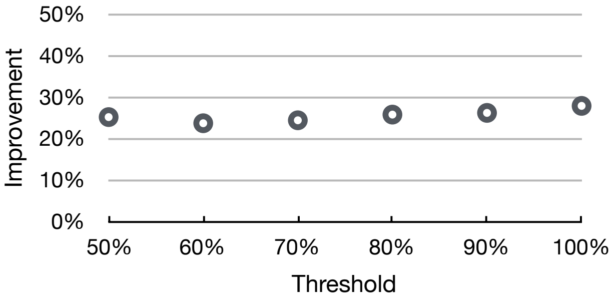


Fig. 7. Improvement of benchmark success rate w.r.t. varying thresholds for defining successful synthesis.

To ensure that our threshold choice of 100% for defining successful benchmark synthesis is not overtly conservative, we investigate the impact of using different thresholds for defining successful benchmark synthesis in Fig. 7. The y-axis tracks the improvement in benchmark success rate of SketchAX I over Sketch for all benchmarks. Notice that (1) the improvement is roughly the same (specifically, between **24%** and **28%**) across all thresholds and (2) the improvement is actually the highest for the 100% threshold (specifically, **28%**). Henceforth, we use 100% as the default threshold to measure benchmark success rate.

Differences in performance over Bit and Int benchmarks. As evident from Fig. 5 and Fig. 6, Sketch has a noticeably higher success rate for Int benchmarks than for Bit benchmarks. In fact, for the Int benchmarks from Sketch Benchmarks Repositories that satisfy some properties, Sketch's instance success rate is 97%![6]

To understand the difference in performance of Sketch for Int and Bit benchmarks, we took a closer look at the approximate sizes of the respective synthesis search spaces and found these to be smaller for Int benchmarks on average. For instance, while none of the Int benchmarks have solution spaces of sizes greater than 10^{50}, 22 out of 111 Bit benchmarks have solution spaces of sizes greater than 10^{50}. The smaller solution space sizes for Int benchmarks are primarily because of the small default bounds used in Sketch for holes corresponding to integer constants, loop iterations etc, and, at least partially, explain Sketch's higher success rate on Int benchmarks.

The difference in the improvement brought by SketchAX I for Int and Bit benchmarks satisfying some properties is harder to exactly pinpoint as there are many factors at play. We believe this is because the partial programs for Int benchmarks already impose significant structural/syntactic constraints on the synthesis search space, thereby limiting the improvements due to the semantic constraints corresponding to relational perturbation properties.

7.2 SketchAX II: Property-Validation UI

Figures 8 and 9 summarize the results of our synthesis experiments with Sketch and all SketchAX algorithms.

Figures 8 and 9 show that SketchAX II, which employs a user to validate 3 perturbed examples per property, has a similar performance as SketchAX I across all categories. Further, observe that for the benchmarks which do not satisfy any properties, SketchAX II performs slightly better than SketchAX I, leading to a small overall improvement in its success rate across all benchmarks. There is an interesting explanation for this. For some synthesis instances, some relational perturbation properties hold on the given example set even though the properties don't hold for the program in general. Thus, the user validates the resulting perturbed examples and SketchAX II applies the properties to successfully augment the example set and the synthesis. In contrast, SketchAX I does not apply such properties to the example set and hence, does not augment the synthesis in these cases.

We also tested SketchAX II by having the user validate 1 and 2 perturbed examples per property and found that the results were a bit worse. The improvement in instance success rate over Sketch for all benchmarks was 21% and 22%, respectively, and for benchmarks satisfying some properties was 56% and 59%, respectively.

7.3 SketchAX III: Property-Inference UI

The noteworthy improvements in success rates over Sketch we have discussed so far have been for SketchAX with the Property-Selection and Property-Validation UIs, where the user plays an active role in providing or helping infer an applicable set of relational perturbation properties. The real testament to SketchAX's ability to augment Sketch lies in the success rates of SketchAX III in Figures 8 and 9. SketchAX III performs similar to SketchAX I in most categories, with the same overall improvement in instance success rate over all benchmarks. The reason for the small improvement in success rate of SketchAX III for benchmarks that do not satisfy any properties is the same as that for SketchAX II.

Inference of correct property sets. Since SketchAX III infers relevant property sets to use to augment examples, we examine its *inference accuracy*. This inference accuracy tracks the percentage

[6] For curious readers, Sketch's instance success rate for SyGuS and the manually-constructed Int benchmarks satisfying some properties is 19% and 49%, respectively.

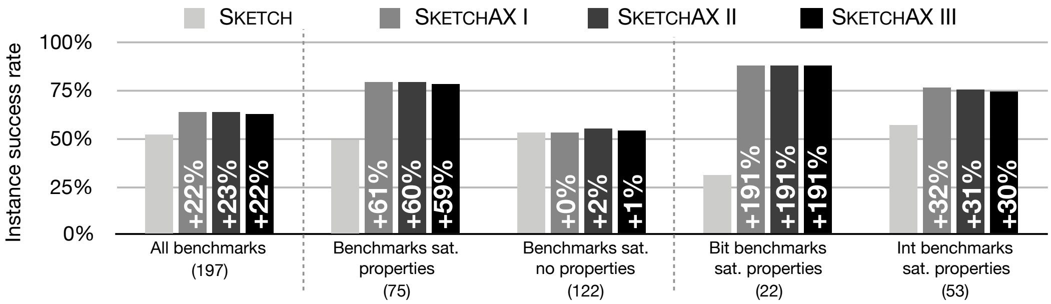


Fig. 8. Instance success rate of Sketch and SketchAX algorithms.

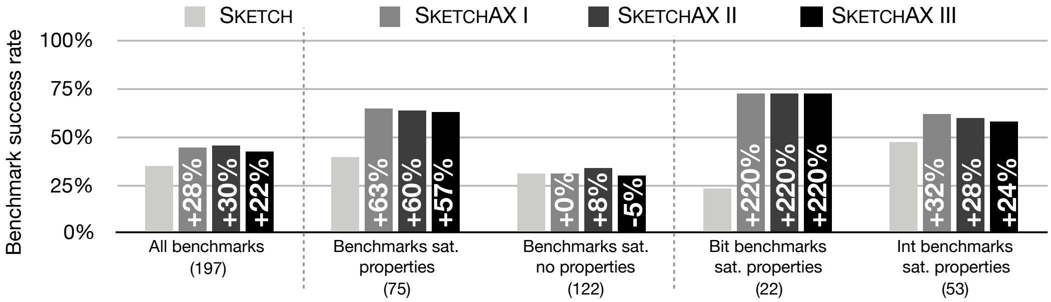


Fig. 9. Benchmark success rate of Sketch and SketchAX algorithms.

of synthesis instances for which SketchAX III inferred property sets that are subsets of the correct property sets. By randomly choosing one of the Max-SMT solutions, SketchAX III's inference accuracy over all benchmarks is 79.4%. The inference accuracy for benchmarks satisfying properties is 92.1% (91.1% for int benchmarks and 94.5% for bit benchmarks). Finally, the inference accuracy for benchmarks that do not satisfy any properties is 71.6% (53.9% for int benchmarks and 78.2% for bit benchmarks).

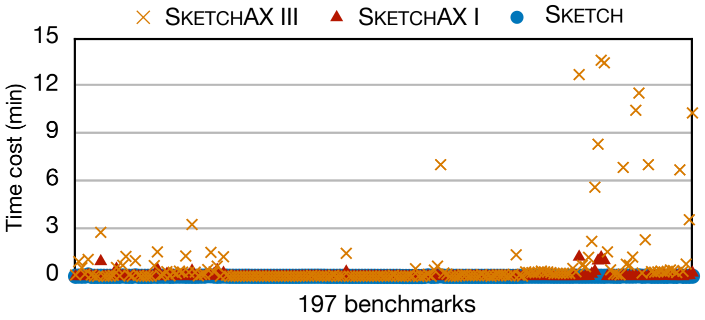


Fig. 10. Time cost of Sketch, SketchAX I and SketchAX III.

7.4 Time Cost of SketchAX

Fig. 10 shows the total synthesis time taken by Sketch and SketchAX I and SketchAX III on each of the 119 test benchmarks. We exclude SketchAX II as, barring the user interactions (whose

Table 2. Statistical view of time cost (in seconds).

	Sketch	SketchAX I	SketchAX III
Average	1.20	4.04	51.56
1st quartile	0.98	1.00	1.43
Median	1.07	1.16	6.03
3rd quartile	1.27	1.70	20.62

time cost is hard to estimate), the computations in SketchAX II are almost identical to those in SketchAX I. Observe that for most benchmarks, SketchAX I and SketchAX III took time similar to that taken by Sketch. The statistical results in Table 2 provide a more fine-grained view of the time performance. While the average times taken by SketchAX I and SketchAX III are noticeably higher than that taken by Sketch, their times for the 3 quartiles are significantly lower and quite reasonable. Notice that SketchAX I closely matches the times taken by Sketch on the 1st and 2nd quartile. And for 75% of the benchmarks (3rd quartile), the time taken by SketchAX I is 1.70 seconds. As for SketchAX III, 25% of the benchmarks (1st quartile) were synthesized within 1.43 seconds and half of the benchmarks (median) were tackled within 6.03 seconds.

7.5 Sensitivity of SketchAX to Size of Example Set

To illustrate how the size of the user-provided example set affects the success rate of SketchAX algorithms, we ran the algorithms with 1 to 5 examples for bit and int benchmarks (see Table 3). As expected, more I/O examples can improve the success rates of all algorithms: (from about **30**% to about **60**% for Sketch and from about **33+**% to about **71**% for SketchAX algorithms). In general, the improvement due to augmented synthesis increases as the number of examples decreases. However, for benchmarks satisfying some relational perturbation properties, the improvement starts going down when the number of examples decreases below a certain threshold (2 in our case). This is because the number of initial examples is too small to reliably augment. The negative improvement cases when using Sketch III for benchmarks satisfying no properties is because we might enforce the wrong properties.

Table 3. Instance success rate (%) with varying example set sizes

	(197) Benchmarks					(75) Benchmarks sat. properties					(122) Benchmarks sat. no properties				
	1 ex.	2 ex.	3 ex.	4 ex.	5 ex.	1 ex.	2 ex.	3 ex.	4 ex.	5 ex.	1 ex.	2 ex.	3 ex.	4 ex.	5 ex.
Sketch	30.0	42.7	52.0	56.3	59.5	31.1	42.3	49.6	52.7	55.9	29.3	43.0	53.4	58.5	61.8
SketchAX I	37.5	53.5	63.5	67.7	70.4	50.8	70.5	79.9	82.7	84.4	29.3	43.0	53.4	58.5	61.8
Improvement	25.0	25.2	22.2	20.3	18.2	63.5	66.9	61.0	57.0	51.1	0.0	0.0	0.0	0.0	0.0
SketchAX II	39.0	54.8	64.1	68.1	70.6	51.2	70.5	79.2	82.3	84.1	31.6	45.2	54.8	59.3	62.2
Improvement	29.9	28.4	23.1	20.9	18.4	64.8	66.9	59.7	56.2	50.2	7.2	5.2	2.3	1.4	0.7
SketchAX III	33.7	52.7	63.4	67.5	70.5	44.7	68.5	78.7	81.6	83.7	26.9	43.0	53.9	58.9	62.3
Improvement	12.2	23.1	21.9	19.9	18.3	43.8	62.1	58.6	54.9	49.9	-8.4	-0.4	0.9	0.6	0.8

7.6 Discussion

We wrap up this section with a discussion of some of our choices.

Sketch. Recall that all the algorithms presented in Sec. 5 and Sec. 6, except for Algo. 4, are parameterized by a synthesizer and can potentially be used with any example-based synthesizer, with support for a deterministic mode and the variable domains described in Sec. 3 and Sec. 4. Hence, we experimented with two other synthesizers that were compatible with our framework —

DryadSynth and CVC4 (the winners in the CLIA Track of SyGuS 2018). Augmented versions of both synthesizers took more than one hour to synthesize a program which returns the maximum of three integers (with 69 perturbed and 3 original examples). Ultimately, we chose Sketch as it was significantly more efficient than these synthesizers, especially when given many augmented I/O examples.

Property inference using a partial Max-SMT formulation. Our approach to infer applicable property sets using a partial Max-SMT formulation (or its greedy implementation) appears to bias the synthesizer towards programs that satisfy relational perturbation properties.

There is adequate empirical evidence that this is not the case for our current experimental setup. Note that 122/197 (62%) benchmarks in our dataset do not satisfy any properties. The instance success rate of SketchAX III for the benchmarks that do not satisfy any properties is similar to the instance success rate of Sketch (see Fig. 8).

We probed further and implemented another version of SketchAX III that explicitly attempts to infer relational perturbation properties that are *not applicable* for a benchmark, in addition to inferring applicable properties. The procedure is initialized with the following set of properties $\mathcal{R} = \{p_1, \neg p_1, p_2, \neg p_2, \ldots, p_8, \neg p_8\}$; for each "negative" property of the form $\neg p$, E_{pert} is generated as before and the corresponding soft constraint is generated as $\bigvee_{(x,y)\in E_{pert}} (P(x) \neq y)$. However, we found that the difference in the performance of SketchAX III and this modified version of SketchAX III is negligible.

All of this indicates that our Max-SMT formulation is not inaccurately biasing the synthesizer towards programs that satisfy some relational perturbation properties. We believe the reason for this is that the inherent bias placed on the search space by the partial programs used for all our benchmarks disallows inference of inapplicable properties (because of some detected inconsistency between augmented examples and original examples/partial program). In the future, as we generalize our approach to other domains and synthesizers, we may need to develop other versions of SketchAX III (for instance, similar to the one outlined above and, perhaps, in combination with one of the ranking functions we experimented with that can order property sets by their applicability to the domain of interest).

Example augmentation with relational perturbation properties. We outline the reasons for enforcing relational properties using augmented I/O examples instead of formal specifications. Another option for augmenting I/O examples is to use a complete functional specification or reference implementation to generate more I/O examples. We did some experiments and found that Sketch can match the performance of SketchAX I with 20-30 random examples for some benchmarks. This is not surprising as relational perturbation properties are not inherently more helpful than the actual functional specification for example augmentation or for placing a semantic bias on the solution space. Of course, in real-world PBE settings, it is challenging to randomly generate a large number of correct I/O examples (as one does not have a complete functional specification or reference implementation), or, expect users to provide large numbers of examples. Thus, example augmentation using relational perturbation properties is essential for ensuring the usability of our technique.

8 RELATED WORK

Data augmentation in machine learning. Deep learning techniques use data augmentation as a common technique for improving machine learning classifiers [Krizhevsky et al. 2012; Simard et al. 2003]. For instance, for learning a classifier on a set of input images, they generate new images by applying label-preserving transformations (e.g. image translation, horizontal reflections, or altering RGB channel intensities). This not only provides more training data for large deep learning models, but also enables the model to learn certain input invariance properties, thereby improving generalization on unseen data. In contrast, our algorithms for example-based synthesis learn the desired program from a small set of examples. Instead of limiting ourselves only to label-preserving transformations, our framework also supports perturbations where the labels (outputs) can change with respect to the input changes, e.g. the permutation preservation and value preservation perturbations.

Metamorphic relations in testing. In software engineering, metamorphic testing uses *metamorphic relations* to help establish a test oracle (see, e.g., [Chen et al. 2018] for a recent review). Since many computations do not have an easily computable reference output (test oracle), the testing framework instead relies on quantifying the relative changes in the inputs and the outputs, defined by the metamorphic relations, to indicate a potential bug. Metamorphic relations typically need to be specified manually by the developer (similar to our Property-Selection UI). It is only recently that researchers have proposed automating inference of specific metamorphic relations for certain application domains, using machine learning [Kanewala et al. 2016; Zhang et al. 2014]. While our relational perturbation properties bear some similarity to metamorphic relations, our approach is the first to leverage relational properties to address the ambiguity/generalizability problem in example-based program synthesis. Moreover, our approach lets a developer interactively identify the properties (through the Property-Validation UI) and automatically infer relevant relational properties using a Partial Max-SMT-based formulation (through the Property-Inference UI).

Handling ambiguity in example-based synthesis. Besides using highly structured DSLs [Alur et al. 2013; Solar-Lezama et al. 2006] to place a syntactic bias on the hypothesis space, many example-based synthesizers use a ranking function that aims to score consistent programs by their ability to generalize. The ranking function is either manually designed using a set of custom weights assigned to different DSL operators [Gulwani et al. 2012] or learnt from data using supervised machine learning techniques [Singh and Gulwani 2015]. Another approach gathers additional information from the user to disambiguate the program space [Mayer et al. 2015], e.g., by creating distinguishing inputs [Jha et al. 2010] or abstract examples [Drachsler-Cohen et al. 2017]. Raychev et al. [2016] present a feedback loop that identifies and discards potentially incorrect examples. Unlike these approaches, our approach handles ambiguity by placing a semantic bias on the hypothesis space using relational perturbation properties to automatically augment the example sets. Our approach is complementary to previous techniques and it might be interesting to investigate combining these techniques in future.

Programming by examples (PBE). PBE [Lieberman 2001] techniques have been successfully developed for various domains: string transformations [Gulwani et al. 2012; Singh 2016], SQL queries [Wang et al. 2017], data structure manipulations [Feser et al. 2015; Singh and Solar-Lezama 2011], number transformations [Singh and Gulwani 2012], parser synthesis [Leung et al. 2015], map-reduce style distributed programs [Smith and Albarghouthi 2016], web data integrations [Inala and Singh 2017]. More recently, there are efforts to use deep learning [Balog et al. 2017; Devlin et al. 2017; Parisotto et al. 2017] to automatically generate PBE systems. Most of these PBE systems generate programs that are consistent with a user-provided set of examples. Systems such as BlinkFill [Singh 2016] also take into account additional specifications (besides examples) from spreadsheets. Our approach can potentially complement some existing synthesizers given corresponding perturbation properties in different domains.

PBE techniques that account for error or noise in examples are somewhat limited [Devlin et al. 2017; Raychev et al. 2016]. Nevertheless, it is an important research direction for PBE and especially for our work since we additionally infer applicable properties from the user-provided examples.

Program synthesis. The field of program synthesis has seen a recent resurgence because of the advancements in search methods and compute hardware [Alur et al. 2013; Gulwani et al. 2017]. The general synthesis problem aims at learning programs from many different forms of specifications (including reference implementations, logical specifications, natural language, examples etc.), whereas in this work we focus on synthesis from input-output examples. Synthesis frameworks based on other incomplete specification mechanisms can potentially benefit from a similar augmentation of the specification with additional perturbation properties.

Relational program synthesis. Recent work on relational program synthesis [Wang et al. 2018] seeks to synthesize programs from complete relational specifications. In contrast, we use a class of relational properties to augment program synthesis from incomplete example-based specifications.

9 CONCLUSION

We proposed a new approach to address the ambiguity/generalizability issue in example-based synthesis based on the idea of example augmentation using relational perturbation properties. We presented solutions for three user interfaces and demonstrated the effectiveness of our approach in significantly boosting the performance of the Sketch synthesizer. Given this proof-of-concept, we plan to explore several future directions. We will investigate richer classes of relational properties in diverse domains and apply our approach to multiple example-based synthesizers. We also plan to work on designing new search algorithms for program synthesis based on relational properties.

ACKNOWLEDGMENTS

We are grateful to Armando Solar-Lezama for his insightful comments at different stages of this work. We also thank our shepherd, Eran Yahav, and the anonymous reviewers for their feedback and guidance in improving this paper. This material is based upon work supported, in part, by the National Science Foundation under Grant No. 1846327 and Grant No. 1846354 and by grants from the Purdue Research Foundation and the Purdue University Integrated Data Science Initiative. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

2019. Syntax-Guided Synthesis Competition. https://sygus.org/.

2019. Sketch Benchmarks Repositories. https://bitbucket.org/gatoatigrado/sketch-frontend/src.

Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-Guided Synthesis. In *Formal Methods in Computer-Aided Design (FMCAD)*. 1–8.

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *International Conference on Learning Representations (ICLR)*.

Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *Comput. Surveys* 51, 1 (2018), 4:1–4:27.

Alessandro Cimatti, Anders Franzén, Alberto Griggio, Roberto Sebastiani, and Cristian Stenico. 2010. Satisfiability Modulo the Theory of Costs: Foundations and Applications. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 99–113.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. RobustFill: Neural Program Learning under Noisy I/O. In *International Conference on Machine Learning (ICML)*. 990–998.

Dana Drachsler-Cohen, Sharon Shoham, and Eran Yahav. 2017. Synthesis with Abstract Examples. In *Computer Aided Verification (CAV)*. 254–278.

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. In *Programming Language Design and Implementation (PLDI)*. 229–239.

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets using Input-Output Examples. In *Principles of Programming Languages (POPL)*. 317–330.

Sumit Gulwani. 2016. Programming by Examples: Applications, Algorithms, and Ambiguity Resolution. In *International Joint Conference on Automated Reasoning (IJCAR)*. 9–14.

Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet Data Manipulation using Examples. *Commun. ACM* 55, 8 (2012), 97–105.

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4, 1-2 (2017), 1–119.

Jeevana Priya Inala and Rishabh Singh. 2017. WebRelate: Integrating Web Data with Spreadsheets using Examples. *Proceedings of the ACM on Programming Languages (PACMPL)* 2, POPL (2017), 2:1–2:28.

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided Component-based Program Synthesis. In *International Conference on Software Engineering (ICSE)*. 215–224.

Upulee Kanewala, James M Bieman, and Asa Ben-Hur. 2016. Predicting Metamorphic Relations for Testing Scientific Software: A Machine Learning Approach using Graph Kernels. *Software Testing, Verification & Reliability* 26, 3 (2016), 245–269.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems (NIPS) 25*. 1097–1105.

Alan Leung, John Sarracino, and Sorin Lerner. 2015. Interactive Parser Synthesis by Example. In *Programming Language Design and Implementation (PLDI)*. 565–574.

Henry Lieberman. 2000. Programming by Example: Introduction. *Commun. ACM* 43, 3 (2000), 72–74.

Henry Lieberman. 2001. *Your Wish is My Command: Programming by Example.* Morgan Kaufmann.

Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin G. Zorn, and Sumit Gulwani. 2015. User Interaction Models for Disambiguation in Programming by Example. In *Symposium on User Interface Software & Technology (UIST)*. 291–301.

Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2017. Neuro-Symbolic Program Synthesis. In *International Conference on Learning Representations (ICLR)*.

Hila Peleg, Sharon Shoham, and Eran Yahav. 2018. Programming Not Only by Example. In *International Conference on Software Engineering (ICSE)*. 1114–1124.

Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. 2016. Learning Programs from Noisy Data. In *Principles of Programming Languages (POPL)*. 761–774.

Patrice Y. Simard, David Steinkraus, and John C. Platt. 2003. Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis. In *International Conference on Document Analysis and Recognition (ICDAR)*. 958–962.

Rishabh Singh. 2016. BlinkFill: Semi-supervised Programming by Example for Syntactic String Transformations. *Proceedings of the VLDB Endowment* 9, 10 (2016), 816–827.

Rishabh Singh and Sumit Gulwani. 2012. Synthesizing Number Transformations from Input-Output Examples. In *Computer Aided Verification (CAV)*. 634–651.

Rishabh Singh and Sumit Gulwani. 2015. Predicting a Correct Program in Programming by Example. In *Computer Aided Verificationi (CAV)*. 398–414.

Rishabh Singh and Armando Solar-Lezama. 2011. Synthesizing Data Structure Manipulations from Storyboards. In *Foundations of Software Engineeringi (ESEC/FSE)*. 289–299.

Calvin Smith and Aws Albarghouthi. 2016. MapReduce Program Synthesis. In *Programming Language Design and Implementation (PLDI)*. 326–340.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 404–415.

Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017. Synthesizing Highly Expressive SQL Queries from Input-Output Examples. In *Programming Language Design and Implementation (PLDI)*. 452–466.

Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2018. Relational Program Synthesis. *Proceedings of the ACM on Programming Languages (PACMPL)* 2, OOPSLA (2018), 155:1–155:27.

Jie Zhang, Junjie Chen, Dan Hao, Yingfei Xiong, Bing Xie, Lu Zhang, and Hong Mei. 2014. Search-based Inference of Polynomial Metamorphic Relations. In *Automated Software Engineering (ASE)*. 701–712.