Program Synthesis with Formal Methods for the OutSystems Language

Rodrigo André Moreira Bernardo
rodrigo.bernardo@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

May 2019

Abstract

Program synthesis is the problem of automatically generating concrete program implementations from high-level specifications that define user intent. OutSystems is a low-code platform for rapid application development, and easy integration with existing systems, featuring both visual and textual programming (expressions). In this work, we focus on the textual programming side. We tackle the problem of synthesizing OutSystems expressions in a setting where the specifications are given in the form of input-output examples, focusing on expressions that manipulate the Integer and Text datatypes. We surveyed the state of the art in program synthesis, and implemented two component-based PBE synthesizers based on an OGIS architecture. Both synthesizers employ a mixture of constraint solving with basic enumerative search, differing on the amount of work they put on the constraint solving phase. We benchmarked both synthesizers and compared them to SyPet [10] on a set of real world problems provided by OutSystems, showing good results for programs of up to size 4.

Keywords: Program Synthesis, Programming by Examples, Component-Based Synthesis, Constraint Solving, Satisfiability Modulo Theories

1. Introduction

Program synthesis is the problem of automatically generating program *implementations* from high-level *specifications*. Amir Pnuelli, former computer scientist and Turing Award, described it as "one of the most central problems in the theory of programming" [26], and it has been portrayed as one of the holy grails of computer science [34, 15]. It is easy to understand why: if only we could tell the computer *what to do* and let it figure out *how to do* it, the task of programming would be so much easier! However, Program synthesis is a very hard problem. Programming is a task that is hard for humans and, given its generality, there is no reason to believe it should be any easier for computers. Computers lack algorithmic insight and domain expertise. Moreover, the challenge is actually twofold: we need to find out both how to tackle the intractability of the program space, and how to accurately capture user intent [15].

It is important to have in mind that program synthesis is not a panacea for solving problems in computer programming. For example, we might be interested in other properties besides functional correctness, such as efficiency or succinctness of the generated program. Also, in the general case, it is impossible for program synthesis to eliminate all sources of bugs. Particularly, it cannot solve problems originating from bad specifications that come from a poor understanding of the problem domain.

Writing specifications is, indeed, a delicate process. This might be better understood with a simple, yet non-trivial, example.

1.1. Example: Sorting

In order to exemplify how the interaction between the user and the computer (from now on referred to as the "synthesizer") might occur, let us suppose we are interested in developing a sorting procedure, `sort`, for lists of integers:

```
sort: (xs: List Int) →(xs': List Int)
sort([])   = []
sort(x::xs) = insert(x, sort(xs))

insert: (x: Int, xs: List Int)
     → (xs': List Int)
insert(x, [])  = [x]
insert(x, y::ys) =
  if x <= y then (x::y::ys)
```

```
            else y::insert(x, ys)
```

The function `sort` would take a list of integers as input and return it sorted in ascending order. One approach to implement `sort` is to resort to an auxiliary function, `insert`, taking an element `x` and a list `xs` as inputs, and returning a new list `xs'`. Assuming that `xs` is sorted, `insert` guarantees that `xs'` is also sorted by placing `x` in the "right place". It is easy to see, by induction, that `sort` is correctly defined.

Nevertheless, it would be nicer if we could just give the synthesizer a specification of what it means for a list to be sorted and let it figure out the implementation. For example, the synthesizer could support using type signatures and predicates as specifications. We could also hint the structure of the implementation to the synthesizer by giving a specification for another function, `insert`, which we could believe to be useful to implement `sort`:

```
isSorted: List Int →Bool
isSorted([])      = True
isSorted([x])     = True
isSorted(x::y::ys) = x <= y and
    isSorted(y::ys)

sort: (xs: List Int) →(xs': List Int)
sortSpec: isSorted(xs')

insert: (x: Int, xs: List Int) →(xs':
    List Int)
insertSpec: isSorted(xs) ==>
    isSorted(xs')
```

There is a problem with our specification, however, as there are many unwanted programs that satisfy it. The program that ignores its input and simply outputs the empty list is one such example. The problem is that the specification does not model our intent precisely. It should be clear that the output `xs'` should be in some way related to the input `xs`, but the current specification does not capture that relation. We must require that `xs'` has exactly the same contents as those of `xs`, meaning that every element of `xs` should appear in `xs'` in the same amount. We could express that requirement as a binary predicate `sameContents` over lists (whose implementation is omitted) and add it to specification:

```
sort: (xs: List Int) →(xs': List Int)
sortSpec: isSorted(xs') and
    sameContents(xs, xs')

insert: (x: Int, xs: List Int) →(xs':
    List Int)
insertSpec: isSorted(xs) ==>
  isSorted(xs') and sameContents(xs',
      x::xs)
```

It should be clear by now why writing specifications can be tricky, even in simple cases. What might be less clear is that the specification as it is might still not be fully specified. There are countless other properties that we might want our sorting procedure to satisfy, such as, for example, stability, complexity, or adaptability.

1.2. OutSystems and PBE (Programming by Examples)

This work was done during an internship at OutSystems. [1] OutSystems is also the name of a low-code platform that features visual application development, easy integration with existing systems, and the possibility to add own code when needed. With OutSystems one should be able to build enterprise-grade applications swiftly and with a great degree of automation.[2] To that effect, the kind of programs we are interested in are expressions in the OutSystems language.[3] Thus, we were interested in building a synthesizer for OutSystems expressions that would be *performant* and *easy to use*. This would imply that the synthesis process should finish in a matter of seconds (instead of hours or days), and the synthesizer should have a "push-button"-style interface that does not force the user to acquire new skills. The paradigm in program synthesis that best fits this scenario is called PBE (Programming by Examples). In PBE, the synthesizer should be able to synthesize "correct" programs merely from a small set of input-output examples. By correct we mean that the program matches what the user had in mind.

The most notable success in the area of program synthesis is incorporated in a tool called FlashFill [12]. FlashFill employs PBE technology and is currently integrated in Microsoft Excel. FlashFill is able to synthesize programs that manipulate strings very fast, typically under 10 seconds. However, the programs are written in a particular DSL that does not suit our needs. In particular, it does not seem to exist an obvious way to map the FlashFill DSL to OutSystems expressions. Another notable PBE synthesizer is SyPet [10], which has been applied in the domain of Java programs.

1.3. Contributions

In this work, we surveyed the state of the art in program synthesis, and implemented two component-based PBE synthesizers based on an OGIS architecture. Both synthesizers employ a mixture of constraint solving with basic enumerative search, differing on the amount of work they put on the constraint solving phase, for which they make use of an

[1] https://www.outsystems.com
[2] https://www.outsystems.com/platform/
[3] https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Logic/Expressions/

SMT (Satisfiability Modulo Theories) solver. The components that we used correspond to a small subset of expressions of the OutSystems API that manipulate Integers and Text. However, the synthesizers are not specific to OutSystems expressions and could be instantiated in other domains. We benchmarked both synthesizers and compared them to SyPet [10]

1.4. Document Structure

Section 2 introduces some basic definitions and examples that will be useful later on. Section 5 surveys the related work and provides a brief introduction to the most common techniques in program synthesis, not only in PBE, but in general. Section 6 details the implementation of our contributions. Namely, we implemented two synthesizers for OutSystems expressions based on constraint solving, a concept that we introduce in the sections before. In Section 7 we evaluate and compare the synthesizers presented in Section 6. We compare both synthesizers to each other and to SyPet. Section 8 concludes the work.

2. Preliminaries

This chapter introduces some definitions and examples that will be useful later on. It might be practical to just skim this chapter at first reading, and then come back as needed.

3. Terminology and Basic Definitions

This section covers some essential terminology that is used in the document.

Program synthesis is fundamentally a search problem, thus, for it to be well-defined there must be some *search space* that contains the objects we are searching for. In our case, these objects are programs, so we have a *program space.*

Definition 1 (Program Space). *The set of all programs in a given programming language.*

Programming languages are usually defined by means of a grammar. For our purposes, we are interested in a particular kind of grammar, called a Context-Free Grammar (CFG).

Definition 2 (Context-Free Grammar). *A Context-Free Grammar is a tuple $(\mathcal{V}, \Sigma, R, S)$, where:*

- *$\mathcal{V}$ is a finite set. It denotes the set of non-terminal symbols.*
- *Σ is a finite set. It denotes the set of terminal symbols.*
- *R is a finite relation from $\mathcal{V}$ to $(\mathcal{V} \cup \Sigma)^*$, where the asterisk denotes the Kleene star operation.*
- *S is a non-terminal symbol. It is the start symbol of the grammar.*

Definition 5 in Section 4 provides an example of a CFG.

In our context, we are working in a PBE setting, where the user explains their intent by means of a set of input-output *examples.*

Definition 3 (Input-Output Example). *An input-output example is a pair $(\mathbf{x}, y)$ where $\mathbf{x}$ is a list of inputs $< x_0, x_1, \ldots, x_n >$, and y is the output.*

We say that a program f *satisfies* an input-output example $(\mathbf{x}, y)$ if $f(\mathbf{x}) = y$. We say that a program satisfies a *set* of input-output examples if it satisfies every example in that set.

4. SMT (Satisfiability Modulo Theories)

Many problems in the real world can be modeled in the form of logical formulas. Thus, it is of great interest to have access to efficient off-the-shelf logic engines, usually called *solvers*, for these formulas. The satisfiability problem is the problem of checking if a given logical formula has a solution. SMT solvers check the satisfiability of first-order logic formulas with symbols and operations drawn from *theories*, such as the theory of uninterpreted functions, the theory of strings, or the theory of linear integer arithmetic. SMT solvers have seen a multitude of applications, particularly in problems from artificial intelligence and formal methods, such as program synthesis, or verification. This section gives a short introduction to SMT, and is based on the chapter on SMT of the Handbook of Satisfiability [6].

4.1. Syntax of SMT Formulas

Here we define the language of well-formed SMT formulas. Formulas are composed of symbols and logical connectives over those symbols.

Definition 4 (Signature). *A signature $\Sigma = \Sigma^F \cup \Sigma^P$ is a set of (Σ-)symbols. Σ^F is the set of function symbols, and Σ^P is the set of predicate symbols. Each symbol has an associated arity. A zero-arity symbol x is called a constant symbol if $x \in \Sigma^F$, and is called a propositional symbol if $x \in \Sigma^P$.*

Definition 5 (Terms and Formulas). *A (Σ-)term t is an expression of the form:*

$$t ::= c \mid f(t_1, \ldots, t_n) \mid ite(\phi, t_0, t_1)$$

where $c \in \Sigma^F$ with arity 0, $f \in \Sigma^F$ with arity $n > 0$, and ϕ is a formula. A (Σ-)formula ϕ is an expression of the form:

$$\phi ::= A \mid p(t_1, \ldots, t_n) \mid t_0 = t_1 \mid \bot \mid \top \mid \neg\phi \\ \mid \phi_0 \rightarrow \phi_1 \mid \phi_0 \leftrightarrow \phi_1 \mid \phi_0 \wedge \phi_1 \mid \phi_0 \vee \phi_1 \mid (\exists x.\, \phi_0) \mid (\forall x.\, \phi_0)$$

where $A \in \Sigma^P$ with arity 0, and $p \in \Sigma^P$ with arity $n > 0$.

4.2. Semantics of SMT Formulas

In this section we explore how SMT formulas are given meaning.

Definition 6 (Model). *Given a signature Σ, a (Σ-)model $\mathcal{A}$ for Σ is a tuple $(A, (\_)^{\mathcal{A}})$ where:*

1. *A, called the universe of the model, is a non-empty set;*
2. *$(\_)^{\mathcal{A}}$ is a function with domain Σ, mapping:*
   - *each constant symbol $a \in \Sigma^F$ to an element $a^{\mathcal{A}} \in A$;*
   - *each function symbol $f \in \Sigma^F$ with arity $n > 0$ to a total function $f^{\mathcal{A}} \colon A^n \to A$;*
   - *each propositional symbol $B \in \Sigma^P$ to an element $B^{\mathcal{A}} \in \{\mathbf{true}, \mathbf{false}\}$;*
   - *and each predicate symbol $p \in \Sigma^P$ with arity $n > 0$ to a total predicate $p^{\mathcal{A}} \colon A^n \to \{\mathbf{true}, \mathbf{false}\}$.*

Definition 7 (Interpretation). *Given a model $\mathcal{A} = (A, (\_)^{\mathcal{A}})$ for a signature Σ, an interpretation for $\mathcal{A}$ is a function, also called $(\_)^{\mathcal{A}}$, mapping each Σ-term t to an element $t^{\mathcal{A}} \in A$ and each Σ-formula ϕ to an element $\phi^{\mathcal{A}} \in \{\mathbf{true}, \mathbf{false}\}$, in the following manner:*

1. *$f(t_1, \ldots, t_n)^{\mathcal{A}}$ is mapped to $f^{\mathcal{A}}(t_1^{\mathcal{A}}, \ldots, t_n^{\mathcal{A}})$;*
2. *$p(t_1, \ldots, t_n)^{\mathcal{A}}$ is mapped to $p^{\mathcal{A}}(t_1^{\mathcal{A}}, \ldots, t_n^{\mathcal{A}})$;*
3. *$ite(\phi, t_1, t_2)^{\mathcal{A}}$ is equal to $t_1^{\mathcal{A}}$ if $\phi^{\mathcal{A}}$ is* **true**, *and equal to $t_2^{\mathcal{A}}$ otherwise;*
4. *$\bot^{\mathcal{A}}$ is mapped to* **false**;
5. *$\top^{\mathcal{A}}$ is mapped to* **true**;
6. *$(t_1 = t_2)$ is mapped to* **true** *if $t_1^{\mathcal{A}}$ is equal to $t_2^{\mathcal{A}}$, and is mapped to* **false** *otherwise.*
7. *Σ-symbols are mapped according to the mapping of the model just as before.*

Definition 8 (Satisfiability). *Given a model $\mathcal{A} = (A, (\_)^{\mathcal{A}})$ for a signature Σ, the model $\mathcal{A}$ is said to satisfy a Σ-formula ϕ if and only if $\phi^{\mathcal{A}}$ is* **true**. *The formula ϕ is said to be satisfiable.*

Definition 9 (Theory). *Given a signature Σ, a (Σ-)theory $\mathcal{T}$ for Σ is a non-empty, and possibly infinite, set of models for Σ.*

Definition 10 ($\mathcal{T}$-Satisfiability). *Given a signature Σ and a Σ-theory $\mathcal{T}$, a Σ-formula ϕ is said to be $\mathcal{T}$-satisfiable if and only if (at least) one of the models of $\mathcal{T}$ satisfies ϕ.*

Definition 11 (SMT Problem). *Given a signature Σ and a Σ-theory $\mathcal{T}$, the SMT problem is the problem of determining the $\mathcal{T}$-satisfiability of Σ-formulas.*

5. Background

This section discusses the fundamentals of the field of program synthesis and a broad overview of the state of the art, with a focus on constraint solving and PBE (Programming by Examples). Section 5.1 discusses several ways to describe intent to the synthesizer. Section 5.2 discusses techniques to search the program space for the intended program. **(author?)** presented a broader overview of the field as of 2017 [15].

5.1. Specifications

The first part of solving a program synthesis problem is figuring out how the user will communicate their intention to the synthesizer. An intention is communicated by a *specification*, and may be given in many different ways, including: logical specifications [16], type signatures [23, 11, 27]; syntax-guided methods [2] such as sketches [34], or components [9, 10, 8, 32]; inductive specifications such as input-output examples [11, 13, 22], demonstrations [20], or program traces [19]; or even other programs [35]. The kind of specification should be chosen according to the particular use case and to the background of the user, and it might dictate the type of techniques used to solve the problem (see Section 5.2).

5.1.1. Logical Specifications

Logical specifications are the canonical way of introducing specifications. In the sorting example from the introduction (1.1) we already saw an example of this where the specifications were given as logical pre/post-conditions over the inputs/outputs of the program. In that case, the specifications were written as predicates in the host programming language. Logical specifications may also be given as loop invariants or general assertions in the code, in order to give more hints to the synthesizer. Complete logical specifications are often difficult to write, or expensive to synthesize from. They usually require a great deal of knowledge about the domain of operation, and are typically not suitable for non-technical users.

5.1.2. Syntactic Specifications

A specification can be seen as a constraint over the space of all possible programs. We have already seen one type of specification, logical specifications, which are a kind of *semantic* specifications, meaning that they constrain the program space over *behavior*. Syntactic specifications are a method of specifying intent in program synthesis where a semantic specification is complemented with some form of *syntactic* constraints on the shape that the desired program can take.

Syntactic specifications are typically provided in the form of a CFG [2], or with sketches [34]. These restrictions provide structure to the set of candidate programs, possibly resulting in more efficient search procedures. They can also be used for the purpose of performance optimizations, e.g., by limiting the search space to implementations that only use a limited amount of lines of code. The learned programs also tend to be more readable and explainable.

Sketching The idea of sketching is to provide skeletons of the programs we want to synthesize, called *sketches*, leaving missing details, called *holes*, for the synthesizer to fill. The synthesizer is then directed by the high-level structure of the skeleton while taking care of finding the low-level details according to user-specified assertions. Sketching is an accessible form of program synthesis, as it does not require learning new specification languages and formalizations, allowing the users to use the programming model with which they are already familiarized. This approach was introduced in the SKETCH system by Solar-Lezama [34], which allowed the synthesis of imperative programs in a C-like language.

Component-Based Synthesis In component-based synthesis [32, 8, 9, 10, 17] we are interested in finding a loop-free program made out of a combination of fundamental building blocks called *components*. These components could be, for example, methods in a library API [32, 10], and the way in which they can be combined forms the syntactic specification for the programs we want to find. They may also be supplemented by additional constraints in the form of logical formulas [8].

Syntax-Guided Synthesis The problem of program synthesis with syntactic specifications was generalized and formalized in the work on SyGuS [2].[4] SyGuS is a community effort with the objective of "formulating the core computational problem common to many recent tools for program synthesis in a canonical and logical manner". The input to this problem consists of a background theory, that defines the language, a semantic correctness specification defined by a logical formula in that theory, and a syntactic specification in the form of a CFG. This effort has helped to create a common format for the definition of program synthesis problems and a growing repository of benchmarks. It has also led to the creation of the SyGuS-Comp annual competition.

5.1.3. Inductive Synthesis

Inductive synthesis is an instance of the program synthesis problem where the constraints are underspecified. Sometimes the domain we want to model is complex enough that a complete specification could be as hard to produce as the program itself, or might not even exist. In other cases, we might want the synthesizer to be as easy and intuitive to use as possible for users coming from different backgrounds.

PBE (Programming by Examples) PBE is an instance of inductive synthesis where the specification is given by input-output examples that the desired program must satisfy. Explicitly giving examples can be preferred due to their ease of use, especially by non-programmers, when compared to more technical kinds of specification, such as logical formulas. The examples may be either *positive*, i.e., an example that the desired program must satisfy, or *negative*, i.e., an example that the desired should *not* satisfy. More generally, given some (implicit) input-output example, we may include asserting properties of the output instead of specifying it completely [28]. This can be helpful if it is impractical or impossible to write the output concretely, e.g., if it is infinite.

PBD (Programming by Demonstration) In PBD the user does not write a specification *per se*; instead the synthesizer is given a sequence of transformation steps (a demonstration) on concrete inputs, and uses them to infer the intended program. The program must be general enough to be used with different inputs. PBD can be seen as a refinement of PBE that considers an entire execution trace (i.e., step-by-step instructions of the program behavior on a given input) instead of a single input-output example. It depicts *how* to achieve the corresponding output instead of just specifying *what* it should be.

Though the concept of PBD is easy to understand, the task of the user can be tedious and time-consuming. Therefore, the synthesizer must be able to infer the intended program from a small set of user demonstrations. Ideally, it would also be able to interact effectively and receive feedback from the user. However, the concept might also be interesting when applied to non-interactive contexts, such as *reverse engineering*.

Lau et al. applied PBD to the text-editing domain by implementing SMARTedit, a system that induces repetitive text-editing programs from as few as one or two examples. The system resembles familiar keystroke-based macro interfaces, but it generalizes to a more robust program that is likely to work in more situations [20]. They have also presented a language-neutral framework and an implementation of a system that learns procedural programs from just 5.1 traces on average [19].

[4]`https://sygus.org`

Ambiguity In inductive synthesis the specifications are inherently ambiguous. Therefore, the intended program should not only satisfy the specifications, but also generalize, effectively trying to figure out the user's intention. Typically, two approaches have been used to solve this problem.

The first approach works by *ranking* the set of programs consistent with examples according to their likelihood of being the desired program. This ranking function should allow for the efficient identification of the top-ranked program without having to perform costly enumeration. There have been manual approaches to create such functions, but it is a time-consuming process that requires a lot of domain expertise. It is also a fragile approach because it depends too much on the underlying DSL. Recently, more automated approaches have been proposed [33, 29], usually relying on machine learning techniques. However, as is usual with machine learning techniques, they require large labeled training datasets.

The second approach is called *active learning*, and (usually) relies on interaction between the user and the synthesizer. Typically, this happens by asking the user for (a small set of) additional input-output examples. Another idea, (similar to the one) introduced by **(author?)** [17], works by finding a *distinguishing input*, an input on which two candidate programs differ, and query the user what the expected output should be for the intended program. Other types of active learning exist such as rephrasing the program in natural language, and accepting negative examples [11].

5.1.4. Programs

A program can also be used as a specification and the job of the synthesizer is then to find another program with the same semantics. We might also be interested in programs that behave the same way as the original one, but, for example, are more efficient (according to some metric).

Typically, the original programs are not made to be efficient, but to be easy to read or to prove correct. They may also appear naturally as specifications in certain use cases, such as superoptimization [25], deobfuscation [17], and synthesis of program inverses [35].

5.2. Search Techniques

The second part of solving a program synthesis problem is deciding which search technique to apply in order to find the intended program. First, we want to ensure that the program satisfies the semantic and syntactic specifications. Second, we want to leverage the specifications and the knowledge we have from the problem domain in order to guide the search. Common search techniques are enumerative search [25, 3] (Section 5.2.1), stochastic search [31, 33] (Section 5.2.2), and constraint solving [8, 9, 10] (Section 5.2.3). Modern synthesizers usually apply a combination of those, enabling us to consider frameworks and techniques for structuring their construction, such as CEGIS [34], CEGIS($\mathcal{T}$) [1], and OGIS [18] (Section 5.2.4).

5.2.1. Enumerative Search

In the context of program synthesis, enumerative search consists of enumerating programs by working the intrinsic structure of the program space to guide the search. The programs can be ordered using many different program metrics, the simplest one being program size, and pruned by means of semantic equivalence checks with respect to the specification. Perhaps surprisingly, synthesizers based on enumerative search have been some of the most effective to synthesize short programs in complex program spaces. A reason why is that the search can be precisely tailored for the domain at hand, encoding domain-specific heuristics and case-by-case scenarios that result in highly effective pruning strategies.

In their overview of the field of program synthesis [15], **(author?)** describe some enumerative search algorithms for finding programs in program spaces defined by a CFG (Context-Free Grammar), which we describe next. The algorithms can be generalized to other types of program spaces such as, e.g., sketches.

Algorithm 1: Enumerative Top-Down Tree Search. Adapted from **(author?)**'s overview [15].

```
input : A specification $\phi$ and a CFG $G$
output: A program $p$ in the grammar $G$
        that satisfies $\phi$
begin
    $P \leftarrow$Queue()
    $P' \leftarrow \{S\}$
    while $P \neq \emptyset$ do
        $p \leftarrow$popFirst($P$)
        if $p$ satisfies $\phi$ then
            return $p$
        for $a \in$ nonTerminals($p$) do
            for $b \in \{b|(a,b) \in R\}$ do
                if not subsumed($p[a \to b]$,
                  $P'$) then
                    $P \leftarrow P \cup \{p[a \to b]\}$
                    $P' \leftarrow P' \cup \{p[a \to b]\}$
```

Top-Down Tree Search The first enumerative strategy is the top-down tree search algorithm (Algorithm 1). It takes as input a CFG $G = (V, \Sigma, R, S)$ and a specification ϕ, and works by exploring the derivations of G in a best-first top-down fashion. The algorithm stores the current programs in a priority queue, P, and stores all the programs found so far in the set P'. Both P and P' are initialized with the partial program that corresponds to the start symbol S of G. The algorithm runs until it finds a program p that matches the specification ϕ or there are no more programs waiting in the queue (meaning that the algorithm fails). At every iteration, we take the program p with the highest priority from the queue and check whether it satisfies ϕ. If yes, we return p. Otherwise, the algorithm finds new (possibly partial) programs by applying the production rules of the grammar to p. The program space is pruned in the next step by ignoring programs that are semantically equivalent (with respect to ϕ) to programs already considered in the past (i.e., subsumed within P').

Algorithm 2: Enumerative Bottom-Up Tree Search. Adapted from **(author?)**'s overview [15].

```
input  : A specification $\phi$ and a CFG $G$
output: A program $p$ in the grammar $G$
        that satisfies $\phi$
begin
    $P \leftarrow \emptyset$
    for $progSize = 1, 2, \ldots$ do
        $P' \leftarrow$enumerateExprs($G$, $E$,
          $progSize$)
        for $p \in P'$ do
            if $p$ satisfies $\phi$ then
                return $p$
            if not subsumed($p$, $P$) then
                $P \leftarrow P \cup \{p\}$
```

Bottom-Up Tree Search The bottom-up tree search algorithm (Algorithm 2) is the dual to top-down tree search algorithm. It also takes a CFG $G = (V, \Sigma, R, S)$ and a specification ϕ, and works by exploring the derivations of the grammar in a bottom-up dynamic programming fashion. This strategy has the advantage over the top-down search that (in general) only complete programs may be evaluated for semantic equivalence. The algorithm maintains a set of equivalent expressions, first considering the programs corresponding to leafs of the syntax tree of the grammar G, and then composing them in order to build expressions of increasing complexity, essentially applying the rules of the grammar in the opposite direction.

Bidirectional Tree Search We can see that a top-down tree search starts from a set of input states, while a bottom-up tree search starts from a set of output states. In both approaches the size of the search space grows exponentially with size of the programs. The bidirectional tree search algorithm tries to attenuate this problem by combining the previous two approaches, starting from both a set of input states and a set of output states. It maintains both sets, evolving in the same way as the previous two algorithms, and stops when it finds a state that belongs to both sets in a sort of meet-in-the-middle approach.

5.2.2. Stochastic Search

Stochastic search is an approach to program synthesis where the synthesizer uses probabilistic reasoning to learn a program conditioned by the specification (i.e., the specification induces a probability distribution over the program space).

Typical stochastic synthesis approaches include, for example: genetic programming [37] where a population of programs is repeatedly evolved by application of biological principles (such as natural selection) while optimizing for a given *fitness* function (e.g., the number of input-output examples that are satisfied); neural networks that learn how to reproduce the intended behavior, or that learn actually interpretable programs [24]; learning a distribution (a *guiding function*) over the components of the underlying DSL in order to guide a weighted enumerative search in the direction of a program that is most likely to meet the desired specification [21, 4].

Sampling the Search Space In this section we describe the stochastic synthesizer used by **(author?)** in their syntax-guided synthesis paper [2]. Their synthesizer learns from examples and is adapted from work on superoptimization of loop-free binary programs [31]. Their algorithm uses the Metropolis-Hastings procedure to sample expressions that are more likely to meet the specification. They define a score function, *Score*, that measures the extent which a given program is consistent with the specification. Then they perform a probabilistic walk over the search space while maximizing this score function. The algorithm works by first picking a program p of fixed size n uniformly at random. They then pick a node from its parse tree uniformly at random and consider the subprogram rooted at that node. They then substitute it with another subprogram of same size and type, chosen uniformly at random, obtaining a new program p'. The probability of discarding p for p' is given by the formula $min(1, Score(p')/Score(p))$. It remains to say how to pick the value of n. Typically we do not know the size of the desired program from the start. In order to tackle this problem, they start by fixing its value at $n = 1$ and at each iteration change its value to $min(1, n \pm 1)$ with with some small probability (default is 0.01).

5.2.3. Constraint Solving

Another approach to program synthesis is to reduce the problem to that of constraint solving by the use of off-the-shelf automated constraint solvers [32, 8, 9, 10, 34, 17] (typically SAT or SMT solvers). The idea is to encode the specification in a logical constraint whose solution corresponds to the desired program. **(author?)** [15] illustrate this in a simple way with an example which we show here. This example also serves as a short introduction to the SMT-LIB language [5], a standard language for SMT solvers. Suppose our programs are composed of operations over two input bitvectors, x and y, of length eight:

$$\begin{aligned} \text{program } P &::= plus(E,E) \mid mul(E,E) \mid shl(E,C) \mid shr(E,C) \\ \text{expression } E &::= x \mid C \\ \text{constant } C &::= 00000000_2 \mid 00000001_2 \mid \ldots \mid 11111111_2 \end{aligned}$$

We consider an expression to be either the input variable x or an 8-bit constant. A program consists of additions and multiplications between expressions, or of shift left/right operations over an expression by a constant. We can declare the type of bitvectors of length 8 in SMT-LIB as:

```
(define-sort Bit8 () (_ BitVec 8))
```

To encode the grammar of well-formed programs we first need to introduce the constant symbols `hP`, `hE0`, `hE1`, `c0` and `c1`, as well as the function symbol `prog`:

```
(declare-const hP Int)
(declare-const hE0 Bool)
(declare-const hE1 Bool)
(declare-const c0 Bit8)
(declare-const c1 Bit8)

(define-fun prog ((x Bit8)) Bit8
(let ((left (ite hE0 c0 x))
      (right (ite hE1 c1 x)))
  (ite (= hP 0) (bvadd left right)
  (ite (= hP 1) (bvmul left right)
  (ite (= hP 2) (bvshl left c1)
                (bvlshr left c1))))))
```

The symbol `hP` encodes the choice of which language construct to pick, while the symbols `hE0`, `hE1` encode the choice of whether `left` and `right` are assigned the values of constants (`c0`, `c1`) or the value of the input (`x`). An assignment to these constant symbols corresponds to a valid program in our grammar, if we ensure that `hP` is in a valid range. We can use `assert` to introduce new clauses that must be satisfied:

```
(assert (>= hP 0))
(assert (< hP 4))
```

Finally, we can also use `assert` to encode a semantic specification. For example, suppose we are interested in a bitvector program P that, for all input x, its output is always positive, i.e., $\forall x. P(x) \geq 0$. In SMT-LIB that can be written as:

```
(assert (forall ((x Bit8)) (bvsge (prog
    x) #x00)))
```

This example shows an end-to-end constraint solving approach to program synthesis. However, encoding the problem this way can sometimes be non-trivial or time-consuming. This led to the appearance of the concept of *solver-aided programming*, where programming languages are enlarged with high-level constructs that give the user access to synthesis without having to deal with the constraint solvers directly. For example, **(author?)** describe the SKETCH system as a "compiler [that] relies on a SAT solver to materialize some language constructs". ROSETTE [36] is a framework for developing solver-aided programming languages embedded in Racket that provides constructs not only for synthesis, but also for verification, debugging and angelic execution.

5.2.4. Oracle-Guided Inductive Synthesis

Oracle-guided inductive search (OGIS) is an approach to program synthesis where the synthesizer is split into two components: the *learner* and the *oracle*. The two components communicate in an iterative *query/response* cycle, as shown in Figure 1. The learner implements the search strategy to find the program and is parameterized by some form of semantic and/or syntactic specifications (see 5.1). The usefulness of the oracle is defined by the type of queries it can handle and the properties of its responses. The characteristics of these components are typically imposed by the application.

Typical queries and response types are some of the following [18]:

- *Membership queries*, where given an I/O example x the oracle responds with the answer to whether x is positive or not.
- *Positive (resp. negative) witness queries*, where the oracle responds with a positive (resp. negative) I/O example, if it can find any, or $\bot$ otherwise.

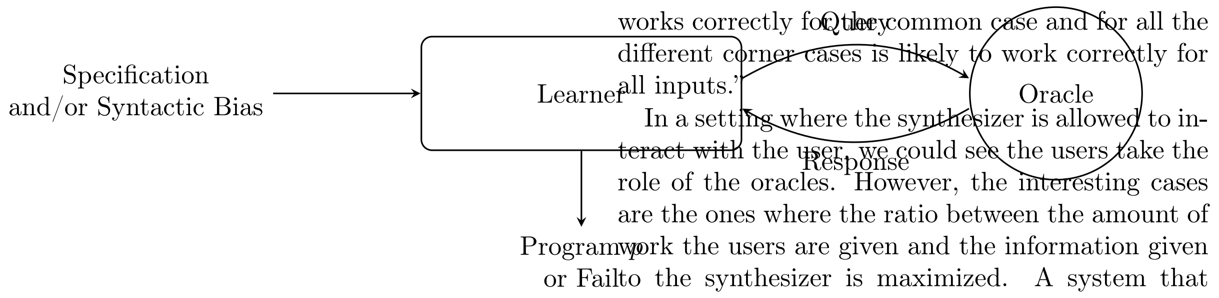

Figure 1: Oracle-guided inductive search. Adapted from **(author?)** [18].

- *Counterexample queries*, where given a candidate program p the oracle responds with a positive I/O counterexample that p does not satisfy, if it can find any, or $\perp$ otherwise.
- *Correctness queries*, where given a candidate program p the oracle responds with the answer to whether p is correct or not. If it is not, the oracle responds with a positive I/O counterexample.
- *Verification queries*, where given program p and specification ϕ the oracle responds with the answer to whether p satisfies ϕ or not, or $\perp$ if it cannot find the answer.
- *Distinguishing input queries*, where given program p and set X of I/O examples that p satisfies the oracle responds with a new program p' and a counterexample x to p that p' satisfies along with all the other examples in X.

An OGIS system responding to counterexample queries corresponds to the *CEGIS* system, introduced by **(author?)** [34] in the context of the SKETCH synthesizer. Correctness oracles are more powerful than counterexample oracles because they are guaranteed to return a counterexample if the program is not correct, where the counterexample oracles might not.

The concepts of OGIS was introduced by **(author?)** [18] as a generalization of CEGIS when they applied this idea to a PBE synthesizer based on distinguishing inputs in order to deobfuscate malware and to generate bit-manipulating programs. Jha et al. further developed this idea by presenting a new theoretical framework for inductive synthesis [18].

In general, the higher the capabilities of the oracles, the more expensive they are to run. Distinguishing oracles are (typically) not as strong as counterexample or correctness oracles as the returned counterexample is not necessarily positive. To understand why they might be effectives tools we can turn to the Bounded Observation Hypothesis [34], which asserts that "an implementation that works correctly for the common case and for all the different corner cases is likely to work correctly for all inputs.'

In a setting where the synthesizer is allowed to interact with the user, we could see the users take the role of the oracles. However, the interesting cases are the ones where the ratio between the amount of work the users are given and the information given to the synthesizer is maximized. A system that frequently queries the users for correctness checks would probably feel very cumbersome. On the other hand, a system that queries for membership or positiveness checks might be more realistic, as usually the user has an idea of what sort of examples fit their desired model.

6. Synthesis

This section describes the problem (Section 6.1) and the approaches that were studied in order to solve it. The first one (Section 6.2), is an adaptation of a component-based constraint solving approach previously applied to the synthesis of bitvector programs [14, 17].

6.1. Problem Description

We are working in the context of the OutSystems platform. OutSystems is a low-code platform that features visual application development, easy integration with existing systems, and the possibility to add own code when needed. To that effect, the kind of programs we are interested in are expressions in the OutSystems language [5].

We can think of OutSystems expressions as a simple functional language of operands and operators that compose themselves to create pure, stateless, loopless programs. This means that OutSystems expressions do not have side-effects, like printing to the screen, or writing to a database, and do not permit any variables or (for/while) loops. They do, however, have conditional expressions in the form of "if" statements. The library of builtin expressions includes functions that manipulate builtin data types such a text strings, numbers, or dates [6] [7].

In this work we are mainly interested in synthesizing expressions that manipulate text strings, like concatenation, substring slicing or whitespace trimming. As some of these operations involve indexing, we are also interested in synthesizing simple arithmetic expressions involving addition and subtraction. Therefore, the data types we are working

[5] `https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Logic/Expressions`

[6] `https://success.outsystems.com/Documentation/10/Reference/OutSystems_Language/Data/Data_Types/Available_Data_Types`

[7] `https://success.outsystems.com/Documentation/10/Reference/OutSystems_Language/Logic/Built-in_Functions`

Function Signatures	Description
Concat(t: Text, s: Text): Text	Concatenation of the texts t and s.
Index(t: Text, s: Text, n: Integer): Text	Retrieves first position of s at or after n characters in t. Returns -1 if there are no occurrences of s in t.
Length(t: Text): Integer	Returns the number of characters in t.
Substr(t: Text, i: Integer, n: Integer): Text	Returns the substring of t with n characters starting at index i.
+(x: Integer, y: Integer): Integer	Integer addition.
-(x: Integer, y: Integer): Integer	Integer subtraction.

Table 1: Description of the builtin functions used for synthesis.

Function Signatures	Examples
Concat(t: Text, s: Text): Text	Concat("", "") = "" Concat("x", "yz") = "xyz"
Index(t: Text, s: Text, n: Integer): Text	Index("abcbc", "b", 0) = 1 Index("abcbc", "b", 2) = 3 Index("abcbc", "d", 0) = -1
Length(t: Text): Integer	Length("") = 0 Length("abc") = 3
Substr(t: Text, i: Integer, n: Integer): Text	Substr("abcdef", 2, 3) = "cde" Substr("abcdef", 2, 100) = "cdef"

Table 2: Examples for the builtin functions used for synthesis.

with are text strings and integers. In particular, we are not dealing neither with boolean, nor conditional expressions. Table 1 describes the builtin expressions that our synthesized programs can be composed of. Table 2 shows input-output examples for each of the functions in Table 1.

Example 6.1. *Suppose that, given a text representing an email, we are interested in extracting its domain part. For example, given the text "john.doe@outsystems.com", we would like to obtain "outsystems.com". The following expression satisfies the specification.*

```
prog(email) = Substr(email,
    Add(Index(email, "@", 0), 1),
    Length(email))
```

Example 6.2. *Suppose that, given a text representing a person's name, we are interested in extracting the first name and prepend it with a prefix. For example, given the text "John Michael Doe" and the prefix "Dr. " we would like to obtain "Dr. John". The following expression satisfies the specification.*

```
prog(name, prefix) = Concat(prefix,
    Substr(name, 0, Index(name, " ",
    0)))
```

Example 6.3. *Suppose now that, given a text representing a person's name, we are interested in extracting not the first but the second name. For example, given the name "John Michael Doe" we would like to obtain "Michael". The following expression satisfies the specification.*

```
prog(name) = Substr(name,
  Index(name, " ", 0),
  Index(name, " ",
    Index(name, " ", 0) + 1) -
        Index(name, " ", 0))
```

In our context, we are working in a PBE setting, so we are interested in synthesizing an OutSystems expression from a set of input-output examples $E = \{(I_i, o_i)\}_i$. For example, `{(<"John Michael Doe">, "Michael")}` is a set of input-output examples that are satisfied by the previous program. However, in this case, the program `prog(name)= Substr(0, 5, 7)` would have done just fine. In order to rule out this program, we could extend the input-output examples with a second example, for instance, `(<"Anne Marie Joe">, "Marie")`.

6.2. Setwise Encoding

Because OutSystems expressions are composed of self-contained pure functions, this synthesis problem fits nicely in the component-based synthesis

```
prog(name, prefix):
  c0 = " "
  c1 = 0
  r1 = Index(name, c0, c1)
  r2 = Substr(name, c1, r1)
  r3 = Concat(prefix, r2)
```

Figure 2: SSA representation of the program from Example 6.2. The last variable, `r3`, is assumed here to be the return value of the program.

paradigm (5.1.2). Therefore, assume we are given a *library* of base components F that the synthesizer can use in order to compose the programs. These components will be builtin functions drawn from the OutSystems library, or combinations of them. Each component can take a finite number of inputs and return exactly one output. More formally, a component $f \in F$ is represented by an expression ϕ_f (a specification) that specifies how its input parameters P_f relate to its return value r_f.

We can see from the previous examples that OutSystems expressions can also include constant literals, like `" "`, or `0`. These could have be given as input, but we would like them to be figured out automatically by the synthesizer. Ignoring well-typedness, an OutSystems expression is a tree-like program whose form can be succintly described using a CFG:

$$S ::= f(S, \dots, S) \mid x \mid c$$

where $f \in F$, $x \in I$, and $c \in C$, and where I and C are the set of inputs of the program, and the set of constant literals in the OutSystems language, respectively.

It is useful to reason about OutSystems expressions in another representation, called SSA form. A program in SSA form is a line program where every variable is assigned exactly once and defined before it is used. For example, the program from Example 6.2 could be written in SSA form as shown in Listing 2. The body of a program in this format can be described succinctly with the following CFG:

$$S ::= ID = c \mid ID = f(x_1, ..., x_n) \mid S; S$$

ID stands for an identifier in the OutSystems language. The non-terminal S represents a line in the program. A line is an assignment of a variable to a constant literal c or to the return value of a component f on inputs $x_1, ..., x_n$. As long as the program is well-typed, an input to a component can be one of the inputs of the program, or variable defined in a preceding line. Thus, the general structure of a program in SSA form is a sequence of assignments.

The approach described in this section is based on **(author?)**'s program encoding. The idea is to

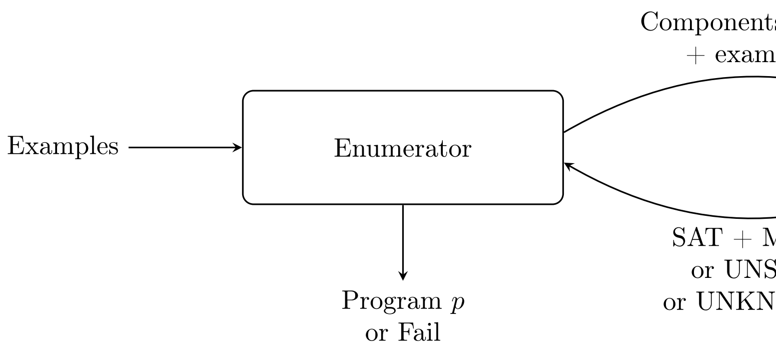

Figure 3: Diagram of the synthesizer.

encode the program space in a formula. The formula is then constrained further in order to encode only those programs that satisfy the input-output examples. A solution to the formula can then be decoded back yielding a program that satisfies the set of examples.

The synthesizer (Figure 3) follows the OGIS model, described in Section 5.2.4. It has a learner part and an oracle part, which will call here the *enumerator* and the *solver*, respectively.

The enumerator receives the set of input-output examples as input, and is parameterized by the library of components. The enumerator is responsible for drawing a subset of components from the library. The components are drawn by trying all sets of combinations (with replacement) in order of increasing size. It then passes these components to the solver, along with the input-output examples, and queries whether there is any program made only of those components that satisfies the examples. There is the additional restriction that the program must use each of the components in the query exactly once.

The solver works by encoding the query into an SMT formula, and uses an automated SMT solver to check for satisfiability. The SMT solver might or might not be able to solve the formula. If the formula is satisfiable, the solver responds to the enumerator with SAT and a solution (called a *model*) to that formula. If not, it responds UNSAT or UNKNOWN, depending on whether the formula is unsatisfiable or the SMT solver could not verify its satisfiability, respectively.

The procedure keeps going in a loop until the enumerator receives SAT from the solver. The enumerator then decodes the model into an actual program, which is then returned.

6.2.1. Program Formula

Let us take the program from Figure 2 as a running example in order to understand how, given the input-output examples and a set of components, we can construct a formula whose model can be de-

```
prog(name, prefix):
  c0 = " "
  c1 = 4
  c2 = 0
  r1 = Concat(prefix, name)
  r2 = Index(r1, c0, c1)
  r3 = Substr(r2, r1, c1)
```

```
prog(name, prefix):
  c0 = " "
  c1 = 0
  r1 = Index(name, c0, c1)
  r2 = Substr(name, c1, r1)
  r3 = Concat(r2, prefix)
```

Figure 4: Two other well-formed programs using the components `Index`, `Substr`, and `Concat`.

coded into a program that satisfies the examples. This program is good because it is a small, non-trivial program, which uses non-input constant variables (`" "` and `0`), and not every component has the same return type.

In order to encode the space of valid programs, the solver has to decide (1) how many constant variables to create and which values to assign them, (2) in which order the components appear in the program, and (3) which *actual* values to pass to the *formal* parameters of each component.

For instance, Figure 4 shows two other valid programs using the components `Index`, `Substr`, and `Concat`. The program on the left does actually satisfy the sole input-output example of Example 6.2, altough it does not generalize. It does so by switching the order of the components, and using one more variable than program 2. The program on the right, however, does not satisfy the example because the values passed to `Concat` are reversed.

In order to encode the program we need variables in the formula to model several entities: (1) the input variables to the program; (2) the constant variables; (3) the formal parameters of each component; (4) the return variables of all components; (5) the output variable of the program; and (6) the *connections* between the variables, that specify which actual parameters are passed to the formal parameters of each component. Thus, we have a set I of input variables, a set C of constant variables, a set P of the formal parameters of all components, a set R of the return variables of all components, and a variable o, the output variable of the program. We will denote the formal parameter variables and return variable of component f by P_f and r_f , respectively. Also, we will use F' to refer to the components available to the solver (in this case, `Index`, `Substr`, and `Concat` — recall that F is used to denote the library of all components). For program 2 this would mean $I = \{name, prefix\}$, $C = \{c_0, c_1\}$, and $R = \{r_1, r_2, r_3\}$. We would also have $P = P_{Index} \cup P_{Substr} \cup P_{Concat}$, with $P_{Index} = \{p_{11}, p_{12}, p_{13}\}$, $P_{Substr} = \{p_{21}, p_{22}, p_{23}\}$, and $P_{Concat} = \{p_{31}, p_{32}\}$.

Well-Formedness Constraint To encode the connections, we require a set L of integer-valued *location* variables l_x for each variable $x \in I \cup C \cup P \cup R \cup \{o\}$. Intuitively, if x is the return variable of component f, then l_x is the line number where f appears in the program. If x is a formal parameter of some component, then l_x is the line number where the actual parameter is defined. In practice, each variable in I is assigned a line number from 1 to $|I|$ (in the obvious way), variables in C are assigned a number from $|I| + 1$ to C, and the output variable $\{o\}$ is assigned the line number $|I| + |C| + |R|$ (the last line). The locations of variables in R range from $|I| + |C| + 1$ to $|I| + |C| + |R|$. The location of each formal parameter $x \in P$ ranges from 1 up to the location of its corresponding component. For program 2 we would have $l_{name} = 1$, $l_{prefix} = 2$, $l_{c_0} = 3$, $l_{c_1} = 4$, and $l_o = 7$. The range constraints are $4 \le l_x \le 7$ for $x \in R$, $1 \le l_x < 5$ for $x \in P_{Index}$, $1 \le l_x < 6$ for $x \in P_{Substr}$, and $1 \le l_x < 7$ for $x \in P_{Concat}$. In general, we can capture these constraints with the following formula:

$$\psi_{\texttt{range}}(I, C, P, R) = \bigwedge_{f \in F'} (|I|+|C|+1 \le l_{r_f} \le |I|+|C|+|R|) \quad \wedge \bigwedge_{f \in F'\, p}$$

The locations of the variables $x \in I \cup C \cup \{o\}$ are known as soon as we decide how many constant variables the program will have at its disposal. The objective is then to find an assignment to the locations of the variables $x \in P \cup R$. These give us all the information we need to decode back the program. For program 2 we have $l_{r_1} = 5$, $l_{r_2} = 6$, $l_{r_3} = 7$; $l_{p_{11}} = 1$, $l_{p_{12}} = 3$, $l_{p_{13}} = 4$; $l_{p_{21}} = 1$, $l_{p_{22}} = 4$, $l_{p_{23}} = 5$; and $l_{p_{31}} = 2$, $l_{p_{32}} = 6$. Because the program has two inputs, we need to subtract two to the location variables to get the corresponding "line numbers". This means, for example, that `Index`, `Substr`, and `Concat` appear on lines 3, 4 and 5, respectively, and so on.

We need a few more constraints in order to encode the space of well-formed programs. First, no two components should have the same location. Thus, we have $l_{r_1} \ne l_{r_2} \ne l_{r_3}$. In the general case:

$$\psi_{\texttt{rloc}}(R) = \bigwedge_{\substack{x,y \in R \\ x \not\equiv y}} (l_x \ne l_y)$$

Second, the program must be well-typed, so the location of each formal parameter $x \in P$ should differ from the location of any $y \in I \cup C \cup R$ whose type does not match with x. In the same vein, only components whose return value has the same type as the output may appear in the last line. These constraints can be written in the following way:

$$\psi_{\texttt{tloc}}(I, o, C, P, R) = \bigwedge_{p \in P} \bigwedge_{\substack{x \in I \cup C \cup R \\ type(p) \neq type(x)}} (l_p \neq l_x) \quad \wedge \bigwedge_{\substack{r \in R \\ type(r) \neq type(o)}} (l_r \neq l_o)$$

Combining formulas $\psi_{\texttt{range}}$, $\psi_{\texttt{rloc}}$, and $\psi_{\texttt{tloc}}$ we get the full program well-formedness constraint:

$$\psi_{\texttt{wfp}}(I, o, C, P, R) = \psi_{\texttt{range}}(I, C, P, R) \wedge \psi_{\texttt{rloc}}(R) \wedge \psi_{\texttt{tloc}}(I, o, C, P, R)$$

Functional Constraint The formula we arrived to in the last section encodes the space of all *syntactically* well-formed programs. However, in no way does it constrain the programs to have the correct *semantics*. In particular, it does not (1) relate the return values to their corresponding components; (2) make sure that variables share the same value if they share the same location; nor (3) make sure that the program satisfies the input-output example. For example, for program 2 we would like to make sure (1) that the value of r_1 is actually equal to `Index(name, c0, c1)`; (2) that p_{21}, the first formal parameter of `Substr`, and $name$ share the same value; and that (3) r_3 equals o, the output variable.

Constraint (1) can be guaranteed by the following formula:

$$\psi_{spec}(P, R) = \bigwedge_{f \in F'} \phi_f(P_f, r_f)$$

Recall that ϕ_f denotes the specification of component f, which relates its formal parameters to its return value. Constraint (2) refers to the dataflow properties of the program. For example, we would like to have the constraints $l_{p_{21}} = l_{prefix} \implies p_{21} = prefix$ (because we do not know beforehand that $p_{21} \neq prefix$), or $l_{r_2} = l_o \implies r_2 = o$ (because we do not know which component is going to be on the last line), but we do not want $l_{p_{21}} = l_{r_1} \implies p_{21} = r_1$ (because p_{21} and r_1 have different types). In general, these properties can be encoded in the following formula:

$$\psi_{flow}(I, C, P, R) = \bigwedge_{p \in P} \bigwedge_{\substack{x \in I \cup C \cup R \\ type(p) = type(x)}} (l_p = l_x \implies p = x) \quad \wedge \bigwedge_{\substack{r \in R \\ type(r) = type(o)}} (l_r = l_o \implies r = o)$$

We would like to make sure that every component given by the enumerator is effectively used in the generated program, meaning that their correspondent return value should be either the actual parameter of some other component, or the final output of the program. This makes sense because of the way that the enumerator draws components from the library (combinations with replacement in order of increasing size), as every subset of F' would have already been passed to the solver and deemed insufficient in order to build a satisfying program. For instance, the return value of `Concat` could be either the output of the program, or one of the actual parameters of the same type of `Input` or `Substr`. Thus, we would have $l_{r_3} = l_o \vee l_{r_3} = l_{p_{11}} \vee l_{r_3} = l_{p_{12}} \vee l_{r_3} = l_{p_{21}}$. In general, this can be encoded in the formula

$$\psi_{out}(o, P, R) = \bigwedge_{f \in F} \bigvee_{\substack{p \in P - P_f \\ type(p) = type(r_f)}} (l_{r_f} = l_p) \quad \vee \bigwedge_{\substack{r \in R \\ type(r) = type(o)}} (l_r = l_o$$

Formula ψ_{flow} along with ψ_{out} guarantee that the generated program has the correct output, thus ensuring that constraint (3) is satisfied. Moreover, we would also like to make sure that no program input goes ignored, significantly cutting down the search space, which can be guaranteed by a formula similar to ψ_{out}:

$$\psi_{in}(I, P) = \bigwedge_{i \in I} \bigvee_{p \in P} (l_i = l_p)$$

The functional constraint is obtained by adding to $\psi_{\texttt{wfp}}$ the formulas from this section, wrapping the formal parameter and return value variables $x \in P \cup R$ under an existential quantifier:

$$\psi_{prog}(I, o, C) = \exists P, R. \, (\psi_{\texttt{wfp}}(I, o, C, P, R) \wedge \psi_{spec}(P, R) \wedge \psi_{flow}(I, C,$$

Full Constraint We are now in position to show the full formula. The formula $\psi_{prog}(I, o)$ encodes a well-formed program that satisfies the input-output example (I, o). We can get a program that works over *all* provided input-output examples $(I, o) \in E$ with a simple conjuction over E like so:

$$\Psi = \exists L, C. \bigwedge_{(I,o) \in E} \psi_{prog}(I, o, C)$$

In essence, this formula encodes the different runs of the program over all the provided input-output examples. A model of this formula corresponds to a program that uses only the components provided by the enumerator, and satisfies all input-output examples. The variables $x \in L \cup C$ should retain their values across all runs, and are the only information we need in order to decode back the program.

6.3. SMT Interlude

At this point it might be nice to sit back and understand how the components of our language (see Table 1) are encoded in SMT. However, it should be noted that the encoding is independent of the particular components used, and works as long as their semantics can be fully encoded in SMT.

In section 4 we introduced the necessary concepts of SMT for the purposes of this thesis. One in particular, was the concept of theory (Definition 9).

Theories constrain the interpretation given to certain symbols. Because we are dealing with text and integers data types, we make use of the SMT theory of strings with linear integer arithmetic. This theory gives a way to reason with symbols representing operations such as integer addition, string length, or substring extraction. The symbols we are interested in are represented in Table 3.

The semantics of components `Concat`, `IndexOf`, `Length`, `Substr`, `Add`, and `Sub` are directly captured by their SMT counterparts $++$, $IndexOf$, $Length$, $Substr$, $+$, and $-$, respectively. Concretely, their specifications are the following:

$$\begin{aligned}\phi_{\texttt{Concat}}(x,y,r) &= (x ++ y = r)\\ \phi_{\texttt{IndexOf}}(s,t,i,r) &= (IndexOf(s,t,i) = r)\\ \phi_{\texttt{Length}}(s,r) &= (Length(s) = r)\\ \phi_{\texttt{Substr}}(s,i,j,r) &= (Substr(s,i,j) = r)\\ \phi_{\texttt{Add}}(x,y,r) &= (x + y = r)\\ \phi_{\texttt{Sub}}(x,y,r) &= (x - y = r)\end{aligned}$$

6.4. Whole Encoding

In the previous approach the workload was split between the enumerator and the solver: the enumerator would exhaustively search for all combinations (with replacement) of the library of components in increasing order of size, while passing them one by one to the solver. In turn, the solver would then verify if there was any way to make a satisfying program using the given components each exactly once. However, the number of combinations of a given size grows exponentially. One wonders if it is possible to absolve the enumerator by pushing as much workload as possible to the solver.

In this next approach we reduce the enumerator to query the solver with a single number n (plus the input-output examples). Instead of drawing components from the library and passing them to the solver, the enumerator now queries if there is any program of exactly n components that satisfies the examples. Thus, the solver is now parameterized by the library of components F. This process is repeated for increasing values of n until a program is found.

6.4.1. Program Formula

Here, we adapt the location-based encoding from the previous section in order to have a solver that finds a program with the exact number of allowed components, n. Just as before, we will have a set I of input variables, an output variable o, a set C of constant variables, a set L of location variables, a set R of return variables, and a set P of formal parameter variables, with some changes, as we will see next.

This time we do not know how many times each component will be used in the synthesized program. This means we must have some way to model the choices of which components get picked and which do not. For this we introduce a new set of integer-valued variables $A = \{a_1, a_2, \ldots, a_n\}$, which we will call the *activation* variables. The activation variables have values in the set $\{1, 2, \ldots, |F|\}$, with the interpretation that if we have $a_i = k$, then component f_k will be the i-th component in the program (the one appearing in line $|I| + |C| + i$). For the same reason this time we do not assign return variables to each component. Instead, as we know the number n of components we are aiming for, we have one return variable r_i, associated with each a_i in the obvious way, for $i = 1, \ldots, n$. These will have their locations fixed beforehand: variable $r_i \in R$ will be assigned to location $l_{r_i} = |I| + |C| + i$.

Example 6.4. *Consider program 2. Assuming that component* ***`Concat`*** *corresponds to $k = 1$,* ***`Index`*** *to $k = 2$, and* ***`Substr`*** *to $k = 3$, then we would have $a_1 = 2$, $a_2 = 3$, and $a_3 = 1$. We would also have $l_{r_1} = 5$, $l_{r_2} = 6$, and $l_{r_3} = 7$.*

As the values of the activation variables $x \in A$ will be found by the solver at constraint-solving time, we also do not know apriori the concrete types of the return variables $x \in R$. Lacking this information, we circumvent the problem by augmenting the type of each return variable to be the union of all possible return types. In our case, this would mean that all return variables may be either of type `Text`, or of type `Integer`.

That is also the case for the types of the formal parameter variables $x \in P$. Moreover, we do not know exactly how many formal parameter variables to create because we do not know the arities of the components that are going to be picked. We know, however, the maximum arity m of all components, so we have an upper bound on the number of formal parameter variables we might need, namely $n * m$. Thus, we introduce variables $p_{ij} \in P$, for $i = 1, 2, \ldots, n$, and $j = 1, 2, \ldots, m$, with the interpretation that p_{ij} is the j-th parameter of f_{a_i}, the component activated by a_i. We also augment their type to be the union of all possible formal parameter types, plus a `Null` type, inhabited by a single value, `null`, to indicate the absence of value. A variable $p_{ij} \in P$ will be `null` if and only if the arity of component f_{a_i} is less than j, and will be ignored by f_{a_i} if that is the case. Finally, we perform the same `Null` type augmentation to the location variables $x \in L$, because there is no sense in having a location number if a variable is null.

Example 6.5. *Consider program 2 and its corresponding symbolic skeleton from Figure 5. We have three components:* ***`Concat`****,* ***`Index`****, and* ***`Substr`****. Components* ***`Index`*** *and* ***`Substr`*** *have the arity equal*

Symbol	Description
$x +\!+\, y$	Function symbol. The terms x and y are strings. Its value, a string, is the string concatenation of x and y.
$IndexOf(s, t, i)$	Function symbol. The terms s and t are strings, and i is an integer. Its value, an integer, is the first occurrence of t in s after the index i, or -1 if t is not in s.
$Length(s)$	Function symbol. The term s is a string. Its value, an integer, is the length (number of characters) of s.
$Substr(s, i, j)$	The term s is a string, and the terms i and j are integers. Its value, a string, is the substring of s starting at index i with length j (or from i until the end of s if the length of s is less than $i + j$).
$x + y$	Function symbol. The terms x and y are integers. Its value, an integer, is the addition of x and y.
$x - y$	Function symbol. The terms x and y are integers. Its value, an integer, is the subtraction of x and y.

Table 3: Some symbols constrained by the theory of strings with linear integer arithmetic.

```
prog($i_0$, $i_1$):
  $c_0$ = ?
  $c_1$ = ?
  $r_1$ = $f_{a_1}$($p_{11}$, $p_{12}$, $p_{13}$)
  $r_2$ = $f_{a_2}$($p_{21}$, $p_{22}$, $p_{23}$)
  $r_3$ = $f_{a_2}$($p_{31}$, $p_{32}$, $p_{33}$)
```

Figure 5: Symbolic representation of a program with two inputs, $i_0, i_1 \in I$ two constant variables $c_0, c_1 \in C$, three return value variables $r_i \in R$, for $i = 1, \ldots, n$, and nine formal parameter variables $p_{ij} \in P$, for $i = 1, \ldots, n$, $j = 1, \ldots, m$, with $n = m = 3$. The question marks '?' are values that will be found by the solver.

to 3, which are the largest among the three components. Thus, we have $n = m = 3$. This means we have nine variables $p_{ij} \in P$. Variables p_{1j}, with $j = 1, \ldots, 3$ are the formal parameters of f_{a_1}, which is component `Concat`. However, `Concat` only takes two parameters. These will be p_{11} and p_{12}, meaning that p_{13} will take value `null`.

Well-Formedness Constraint As with the last encoding, each input variable $x \in I$ is assigned a distinct location l_x from 1 to $|I|$, and each constant variable $x \in C$ is assigned a distinct location from $|I| + 1$ to $|I| + |C|$. As said in the previous section, this time we do the same for each return variable $x \in R$, assigning to each a location from $|I|+|C|+1$ to $|I| + |C| + |R|$. The upper bound on the location (if non-null) of each parameter variable $p_{ij} \in P$ is now statically known to be $|I| + |C| + i$, so now our well-formedness constraint is just:

$$\psi_{\texttt{wfp}}(L, C, I, P) = \bigwedge_{i=1,\ldots,n} \bigwedge_{j=1,\ldots,m} (l_{p_{ij}} \neq \texttt{null} \implies 1 \leq l_{p_{ij}} < |I|+|$$

Compared to the well-formedness constraint of the setwise encoding, this one is arguably made simpler by the fact that the lines of the return variables are fixed apriori, and because we are not doing any type checking.

Example 6.6. *Consider the symbolic skeleton from Figure 5. We have:*

$$\begin{array}{lll} l_{i_0} = 1 & l_{i_1} = 2 & \\ l_{c_0} = 3 & l_{c_1} = 4 & \\ l_{r_1} = 5 & l_{r_2} = 6 & l_{r_3} = 7 \end{array}$$

Also, the well-formedness constraint is the conjunction of the following:

$$\begin{array}{ll} l_{p_{11}} \neq \texttt{null} \implies 1 \leq l_{p_{11}} < 5 & l_{p_{12}} \neq \texttt{null} \implies 1 \leq l_{p_{12}} < 5 \\ l_{p_{21}} \neq \texttt{null} \implies 1 \leq l_{p_{21}} < 6 & l_{p_{23}} \neq \texttt{null} \implies 1 \leq l_{p_{23}} < 6 \\ l_{p_{31}} \neq \texttt{null} \implies 1 \leq l_{p_{31}} < 7 & l_{p_{32}} \neq \texttt{null} \implies 1 \leq l_{p_{32}} < 7 \end{array}$$

Functional Constraint Just as with the last encoding, we have the following concerns regarding the semantics of the program: (1) relating return variables to the values of their components and their parameters; (2) value sharing between variables with the same location; and (3) effectively mapping the inputs $x \in I$ to the output o.

Constraint $\psi_{\texttt{spec}}$, which guarantees constraint (1), should be similar to the one from section 6.2.1, but we will have to pay attention to some special points. First, we have to make sure that, for every component, we relate the correct number of formal parameters to the return value (recall that we may have more formal parameter variables than we may need because we do not know apriori which component goes on which line):

$$\psi_{pnum}(A,P,R) = \bigwedge_{i=1,\dots,n} \bigwedge_{k=1,\dots,|F|} (a_i = k \implies \phi_{f_k}(p_{i1}, p_{i2}, \dots, p_{ij_k}, r_i))$$

In the constraint shown above, j_k is used as shorthand for the arity of component f_k. Second, we should ensure that formal parameter variables that are in excess have value `null`:

$$\psi_{null}(A,P) = \bigwedge_{i=1,\dots,n} \bigwedge_{k=1,\dots,|F|} (a_i = k \implies \bigwedge_{j=j_k+1,\dots,m} p_{ij} = \texttt{null})$$

Finally, we need to constrain the formal parameters to have the correct type, because the specifications ϕ of the components are typed. An example might, perhaps, be the best way to show how this constraint, which we will call ψ_{type}, can be materialized.

Example 6.7. *Consider the symbolic skeleton from Figure 5, and suppose that we are working with components* $f_1 =$ `Concat`*,* $f_2 =$ `Index`*, and* $f_3 =$ `Substr`*. We will use the predicates isstring and isint to mean that a variable must be of type* `Text`*, or* `Integer`*, respectively. For this concrete case, we would the constraint to ensure that the type signatures of these components are satisfied:*

$$\begin{aligned}\psi_{type}(A,P) = & \bigwedge_{i=1,\dots,n} (a_i = 1 \implies (isstring(p_{i1}) \wedge isstring(p_{i2}))) \\ \wedge & \bigwedge_{i=1,\dots,n} (a_i = 2 \implies (isstring(p_{i1}) \wedge isstring(p_{i2}) \wedge isint(p_{i3}))) \\ \wedge & \bigwedge_{i=1,\dots,n} (a_i = 3 \implies (isstring(p_{i1}) \wedge isint(p_{i2}) \wedge isint(p_{i3})))\end{aligned}$$

Constraint $\psi_{\texttt{spec}}$ is then just the conjunction of the three previous constraints:

$$\psi_{\texttt{spec}}(A,P,R) = \psi_{pnum}(A,P,R) \wedge \psi_{null}(A,P) \wedge \psi_{type}(A,P)$$

The dataflow properties of constraint (2) are encoded in a similar way to what we did in section 6.2.1, with the added constraint that every formal parameter variable whose value is `null` must also have a `null` location:

$$\begin{aligned}\psi_{flow}(C,I,P,R) = & \bigwedge_{i=1,\dots,n} \bigwedge_{j=1,\dots,m} \bigwedge_{x \in I \cup C \cup R} (l_{p_{ij}} = l_x \implies p_{ij} = x) \\ \wedge & \bigwedge_{i=1,\dots,n} \bigwedge_{j=1,\dots,m} (l_p = \texttt{null} \implies p = \texttt{null})\end{aligned}$$

Just as last time, we would like to ensure that every return variable is used (as an actual parameter or as the output of the program), a property that be encoded in the following way:

$$\psi_{out}(o,P,R) = \bigwedge_{k=1,\dots,n-1} \bigvee_{i=k+1,\dots,n} \bigvee_{j=1,\dots,m} (l_{r_k} = l_{p_{ij}}) \wedge (r_n = o)$$

Formula ψ_{out}, along with ψ_{flow}, ensures that constraint (3) is satisfied. We would also like to ensure that every input variable appears as an actual parameter of some component, a property encoded by the following formula:

$$\psi_{in}(I,P) = \bigwedge_{x \in I} \bigvee_{i=1,\dots,n} \bigvee_{j=1,\dots,m} (l_x = l_{p_{ij}})$$

The functional constraint is obtained by conjoining the previous constraints with the well-formedness constraint $\psi_{\texttt{wfp}}$, again wrapping the formal parameter and return value variables $x \in P \cup R$ under an existential quantifier:

$$\begin{aligned}\psi_{prog}(A,L,C,I,o) = \exists P,R.\, (\psi_{\texttt{wfp}}(L,P) \wedge \psi_{spec}(A,P,R) \wedge \psi_{flow}(I, \\ \wedge \psi_{out}(o,P,R) \wedge \psi_{in}(I,P))\end{aligned}$$

Full Constraint Just as last time, the functional constraint ψ_{prog} encodes the space of well-formed programs that satisfy a specific input-output example. To obtain a formula that ranges over all input-output examples, we just make a conjuction over the set E:

$$\psi = \exists A, L, C. \bigwedge_{(I,o) \in E} \psi_{prog}(A,L,C,I,o)$$

A model of this formula is a valuation of the variables $x \in A \cup L \cup C$, which can then be used to reconstruct a program that satisfies the input-output examples E using only n components.

7. Experimental Results

In this chapter, we evaluate and compare the synthesizers presented in Chapter ??. The synthesizers are compared in terms of their running time and number of solved problems. A description of the benchmarks is provided in Section 7.1. Section 7.2 presents the results of the benchmarks and a comparison between the setwise synthesizer (Section 6.2) and the whole synthesizer (Section 6.4).

7.1. Benchmark Description

A set of 285522 expressions were provided by OutSystems. We conducted an analysis to determine which builtin functions and which combinations were the most common in that set. We picked 51 expressions containing only those functions (see Table 1) with sizes (number of components) ranging from 1 to 7. A description of these 51 expressions in terms of size is shown in Figure 6.

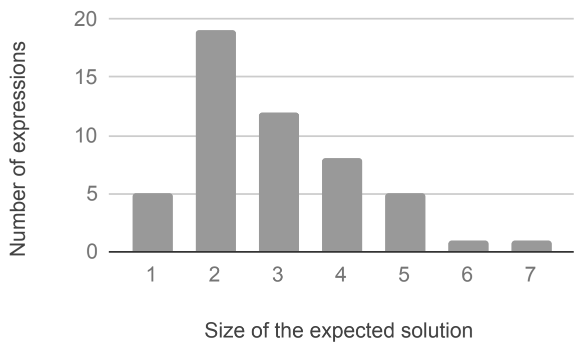

Figure 6: Number of expressions per size. For example, there are 31 expressions of size 2, and 16 expressions of size 1, out of 51 expressions.

The hardness of a benchmark depends on the size of the solution, the number of input-output examples, and the library of components. Typically, the higher the size and the number of components, the harder it is to synthesize a program. Figure 7 shows in how many expressions each component occurs.

We obtained a set of 3 input-output examples for each of these 51 expressions. In order to do that, we developed an *interpreter* for OutSystems expressions, and manually created a set of 3 different inputs for each expression. The inputs were carefully crafted in order to try to eliminate as much ambiguity as possible. Then we interpreted the expressions over their set of inputs in order to obtain the correspondent outputs. A first approach was tried, where we encoded the expressions in SMT in such a way that a solution to the formulas yielded valid input-output examples. Although the approach "worked", it was ultimately too slow, and the automatically generated examples were not natural, so we resorted to the manual approach.

The SMT solver used to solve the formulas generated in the synthesis process was Z3 version 4.8.5 [7]. Each benchmark was monitored by the *runsolver* program [30], and restricted to a wall-clock time limit of 10 minutes, and a memory limit of 16 GB.

7.2. Results

We are interested in the relation between the number and quality of the input-output examples, and their impact on synthesis time and program quality. In this section, we only study the impact of the number of input-output examples.

We present the results for both synthesizers discussed in Chapter **??** with the following configurations. For each instance of 3 input-output examples we ran both synthesizers using 1, 2, or all 3 of the input-output examples. We ran the Set-wise synthesizer (Section 6.2) configured to synthesize programs with a maximum of one integer and one string constants (configurations S-e1:c1, S-e2:c1, and S-e3:c1, for 1, 2, and 3 input-output examples, respectively). We ran the Whole synthesizer (Section 6.4) configured to synthesize programs with a maximum of one constant, which the synthesizer can choose whether it is a an integer or a string (configurations W-e1:c1, W-e2:c1, and W-e3:c1, for for 1, 2, and 3 input-output examples, respectively). Ideally, we would also have tried other configurations to see the impact of using different number of constants. For comparison, we also instantiated SyPet [10], a PBE component-based synthesizer for Java programs that employs a type-directed search together with a constraint-based technique, with our library of components. However, SyPet does not support guessing the value of constants. With this in mind, we set up two configurations for SyPet. For the first configuration (SyPet-All), we introduced a new 0-arity component for each constant in a pool of predefined constants. [8] For the second configuration (SyPet-User), we looked at each particular instance, and introduced new components only for the constants needed for the expected solution. This mimics a setting where the constants are declared by the user, instead of being guessed by the synthesizer. Both Sypet-User and Sypet-All are configured for using all three examples of each benchmark, and may use as many constants as they need (out of the available ones). Table 4 shows a comparison between the different configurations by number of instances solved and running wall-clock time (mean and median for solved instances). An instance is considered

[8]The pool of constants, 43 in total, is the following: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 16, 20, 30, 40, 50, 60, 70, 80, 90, 100, "", "...", "/", "", " ", "-", ".", "@", "," "_","#","(",")","|","<",">",":","Created on ", "updated on", "\", " At:". These were chosen in order to roughly match those found in the expected solutions. The expected solutions employ a total of 34 constants from this pool.`

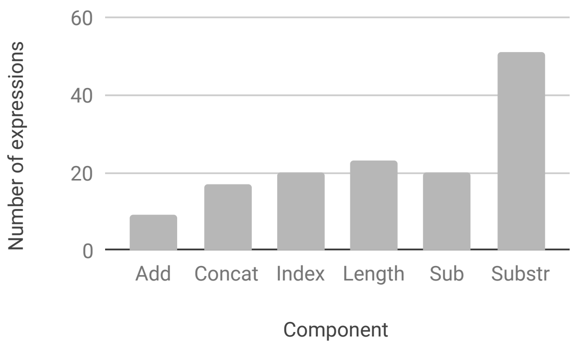

Figure 7: Number of expressions per component.

solved if the synthesized program satisfies the input-output examples (but might or might not generalize). Matching the expected solution means, that the synthesizer outputs a program that, not only satisfies the input-output examples, but also captures the original intent and generalizes to more examples. Figure 8, shows the number of solved instances per size of the expected solution for the configurations S-e1:c1, S-e2:c1, and S-e3:c1. Figure 10 shows the same for configurations W-e1:c1, W-e2:c1, and W-e3:c1. Figure 9 shows the number of solved instances matching the expected solution per size of the expected solution for configurations S-e1:c1, S-e2:c1, and S-e3:c1. Figure 11 shows the same for configurations W-e1:c1, W-e2:c1, and W-e3:c1. Figures 12 and 13 show a comparison between all configurations on three input-output examples.

7.3. Discussion

Both encodings grow linearly with the number of input-output examples. Indeed, we can verify that the hardness of the problem increases with the number of examples (Figures 8, and 10). This effect is more evident the more we increase the size of the expected solution. On the other hand, the number of programs matching the expected solution should increase with the number of input-output examples, which we can also verify (Figures 9 and 11). In Figures 12 and 13 we can see that SyPet-All has similar results to W-e3:c1. The large number of components makes the search space intractable. Besides, it cannot take advantage of its type-directed algorithm because we are dealing with only two types (integers and strings). This last point also applies to SyPet-User, but the advantage given by the user-provided constants proves crucial for its fairly good results, which are actually better than S-e3:c1 (Table 4).

Many expected solutions require more than one constant, with some requiring more than one constant of each type. This helps explain the poor performance of the Whole synthesizer. However, it could be speculated from the start that the Whole synthesizer would be less efficient than the Setwise synthesizer. Typically, solving one large constraint is more expensive than solving multiple smaller constraints. [9] Still, it would be interesting to benchmark configurations with more constants, or with user-provided constants.

The times for the solved benchmarks are reasonably fast, allowing for reasonable interaction times with the user. However, synthesizing programs with 4 or more lines seems to be out of reach for these configurations (Figure 13).

8. Conclusion and Future Work

In this work, we tackle the problem of synthesizing OutSystems expressions from examples, focusing on expressions that manipulate the Integer and Text datatypes. OutSystems sells solutions to help build enterprise-grade applications swiftly and with a great degree of automation. Thus, we were interested in an “push-button”-style approach that would be performant, and could generalize from a small number of examples. We surveyed the state of the art in program synthesis, and implemented two component-based PBE synthesizers – Setwise and Whole. Both synthesizers employ a mixture of constraint solving with basic enumerative search, differing on the amount of work they put on the constraint solving phase. We benchmarked both synthesizers and compared them to SyPet [10] with two different configurations: one where all constants needed to solve the instances are given as new components, and another that simulates user-provided constants by only allowing the constants needed for each particular instance.

The Setwise synthesizer can consistently synthesize programs that satisfy the given input-output with up to four lines, but only manages to match the user intent on programs with up to three lines.

[9] Note that the SMT problem with the theory of strings and linear arithmetic is undecidable.

Configuration	# Instances solved	# Instances solved (matching expected solution)	Mean Time (s)	Median Time (s)
S-e1:c1	48	3	4.62	0.48
S-e2:c1	39	21	51.28	3.21
S-e3:c1	34	23	51.38	4.47
W-e1:c1	47	2	6.76	0.65
W-e2:c1	13	4	11.14	2.24
W-e3:c1	9	5	21.45	7.48
SyPet-All	11	6	31.27	3.00
SyPet-User	32	30	52.75	5.00

Table 4: Comparison between the different configurations by number of instances solved and running wall-clock time for solved instances (not necessarily matching the expected solution).

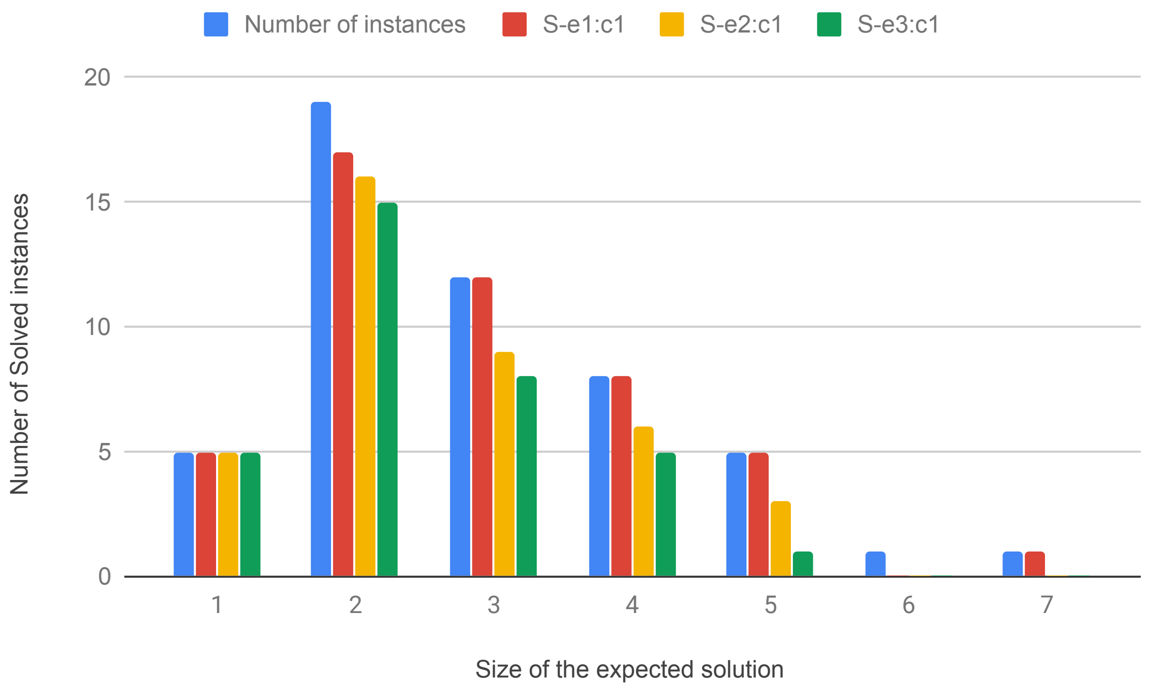

Figure 8: Number of solved instances per size of the expected solution for the Setwise synthesizer with one, two, and three examples, one integer constant, and one string constant (S-e1:c1, S-e2:c1, S-e3:c1, respectively).

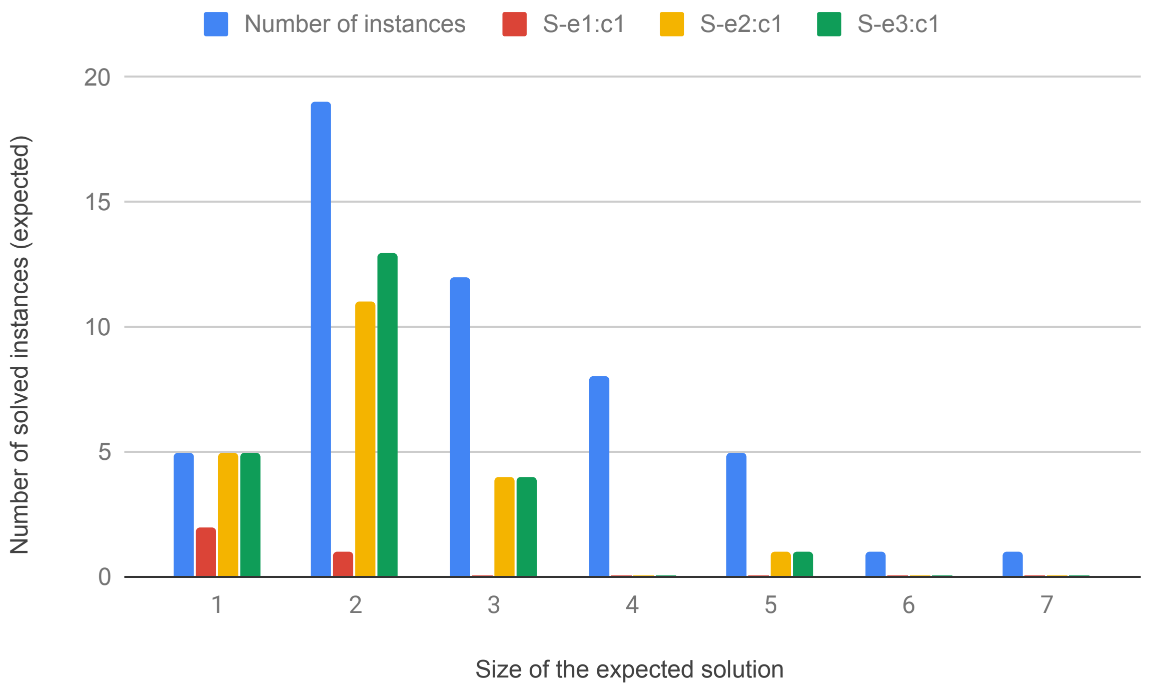

Figure 9: Number of solved instances matching the expected solution per size of the expected solution for the Setwise synthesizer with one, two, and three examples, one integer constant, and one string constant (S-e1:c1, S-e2:c1, S-e3:c1, respectively).

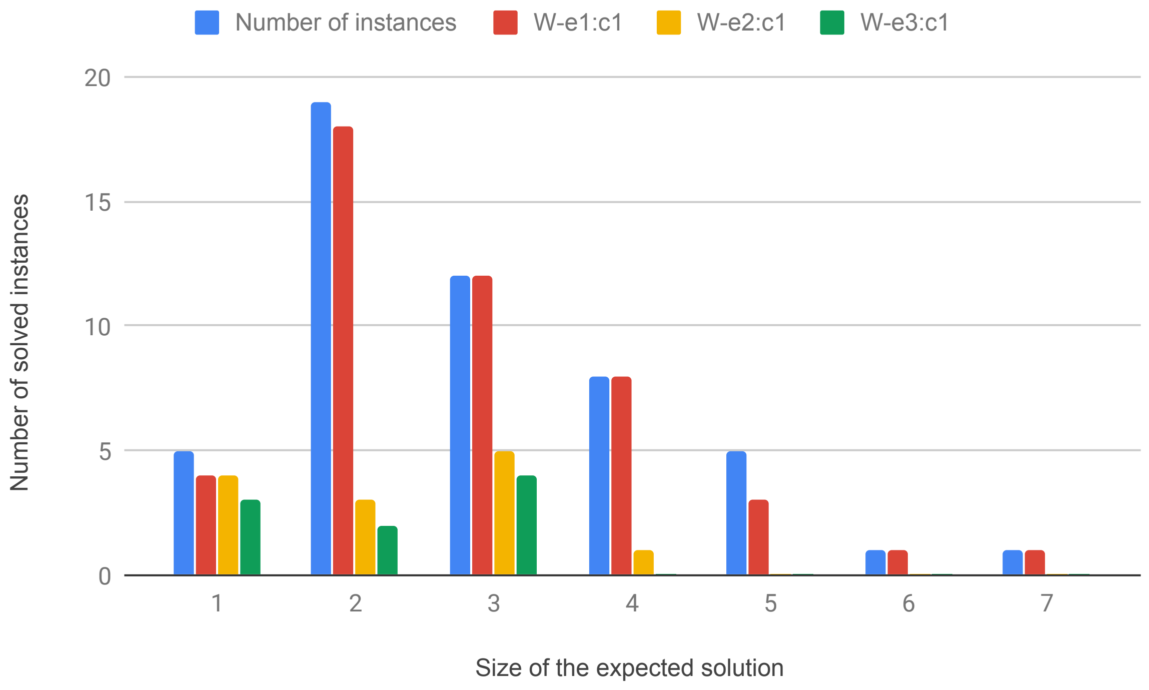

Figure 10: Number of solved instances per size of the expected solution for the Whole synthesizer with one, two, and three examples, and one constant (W-e1:c1, W-e2:c1, W-e3:c1, respectively).

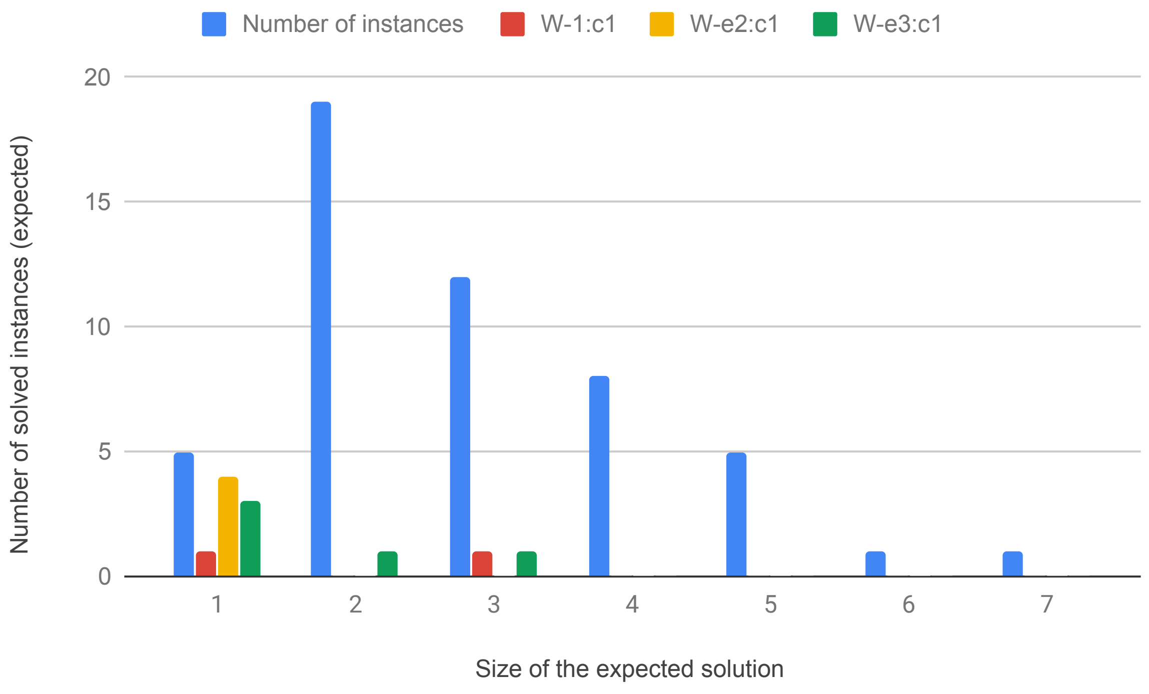

Figure 11: Number of solved instances matching the expected solution per size of the expected solution for the Whole synthesizer with one, two, and three examples, and one constant (W-e1:c1, W-e2:c1, W-e3:c1, respectively).

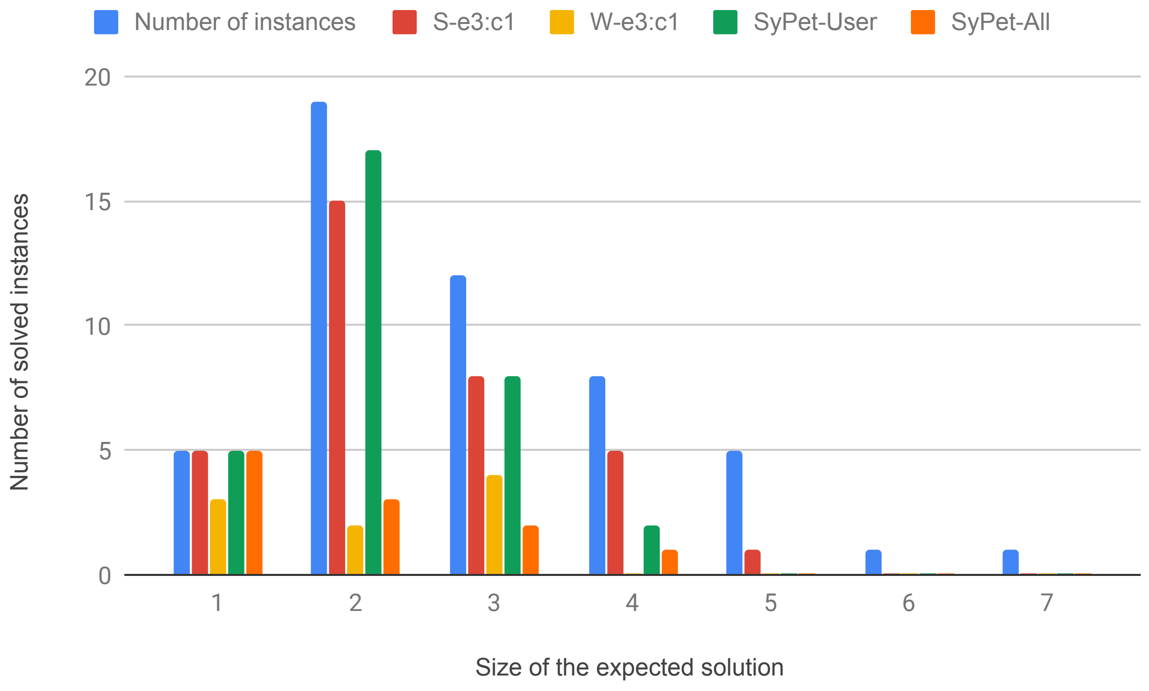

Figure 12: Number of solved instances per size of the expected solution for the Setwise (one integer constant and one string constant) and Whole synthesizers (one constant) with three examples, and both SyPet configurations (user-provided constants and pool of constants).

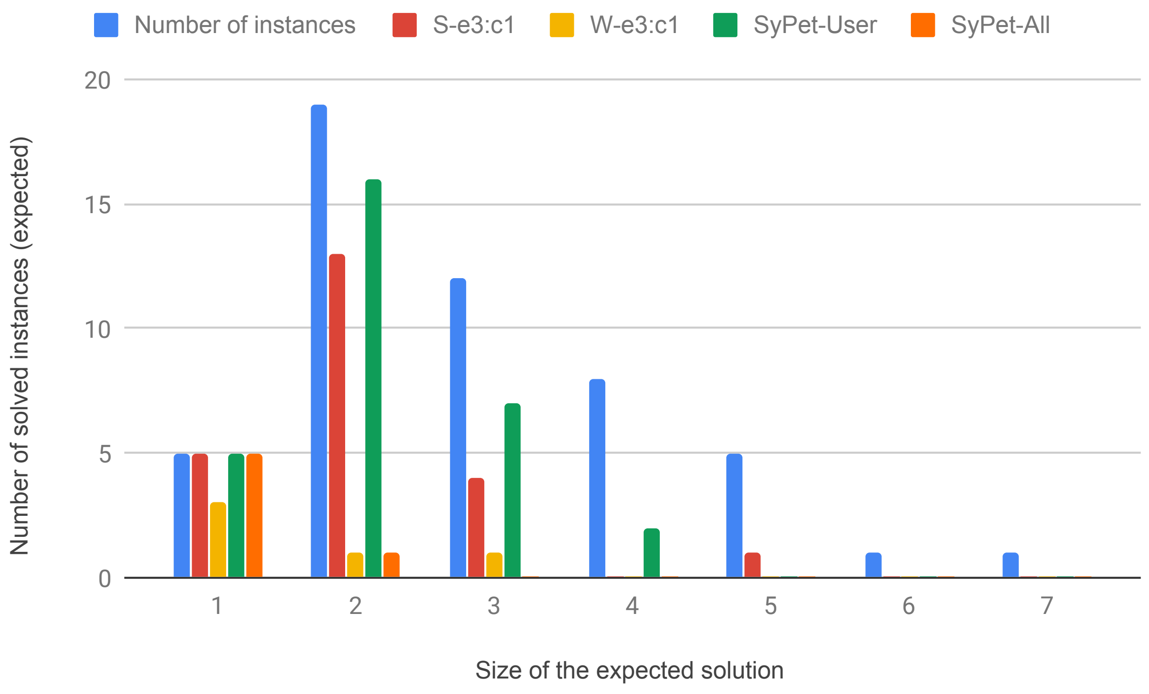

Figure 13: Number of solved instances matching the expected solution per size of the expected solution for the Setwise (one integer constant and one string constant) and Whole synthesizers (one constant) with three examples, and both SyPet configurations (user-provided constants and pool of constants).

Still, it manages to be competitive with SyPet, even when we configured the latter for a scenario simulating user-provided constants. On the other hand, the Whole synthesizer fails to produce good results on both fronts in all but the most trivial instances (programs with one line). However, further benchmarking should be done, and refinements should be applied. In particular, in the future we should benchmark configurations allowing for the use of more constants to properly assess the impact on the number of instances solved and on run time. This should allow for more instances to be solved because it is very common for programs to use more than one or two constants. It would also be interesting to test configurations with user-provided inputs. Another interesting scenario, probably involving extensions to the synthesizers includes trying to figure out from the input-output examples what constants the program might use. Moreover, it would be interesting to see how our synthesizers compare to a synthesizer implemented using the PROSE framework. [10].

Both approaches use encodings whose size scales linearly with the number of input-output examples. While not ideal, it is not unacceptable given that we are interested in offloading as much work as possible from the user, and so the approach should generalize with a small number of examples. Given the experimental results, tending towards approaches more reliant on general-purpose constraint solving might not be the best way to tackle this kind of problems. On the other hand, it would be interesting to explore how solvers specialized for synthesis, or approaches with a tighter integration with the underlying solver might fare. In fact, some approaches in the literature apply this idea [8, 1]. Moreover, it should be explored if this approach scales if we use a larger library of components, especially components that do not have a direct encoding in SMT (and, in turn, could be more difficult, or even impossible to synthesize). It would be particularly interesting to support if-then-else expressions, along with frequently used predicates over Integers and Texts.

[10] `https://microsoft.github.io/prose/`

References

[1] A. Abate, C. David, P. Kesseli, D. Kroening, and E. Polgreen. *Counterexample Guided Inductive Synthesis Modulo Theories: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, pages 270–288. 07 2018.

[2] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh,

A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. IEEE, oct 2013.

[3] R. Alur, A. Radhakrishna, and A. Udupa. Scaling Enumerative Program Synthesis via Divide and Conquer. In *TACAS*, 2017.

[4] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. DeepCoder: Learning to Write Programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[5] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[6] A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications.* IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.

[7] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] Y. Feng, R. Martins, O. Bastani, and I. Dillig. Program Synthesis Using Conflict-driven Learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 420–435, New York, NY, USA, 2018. ACM.

[9] Y. Feng, R. Martins, J. Van Geffen, I. Dillig, and S. Chaudhuri. Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 422–436, New York, NY, USA, 2017. ACM.

[10] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps. Component-based Synthesis for Complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 599–612, New York, NY, USA, 2017. ACM.

[11] J. Frankle, P.-M. Osera, D. Walker, and S. Zdancewic. Example-directed Synthesis: A Type-theoretic Interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 802–815, New York, NY, USA, 2016. ACM.

[12] S. Gulwani. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

[13] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet Data Manipulation Using Examples. *Communications of the ACM*, 55(8):97–105, Aug. 2012.

[14] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of Loop-free Programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 62–73, New York, NY, USA, 2011. ACM.

[15] S. Gulwani, O. Polozov, and R. Singh. Program Synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[16] S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv. A Simple Inductive Synthesis Methodology and Its Applications. *SIGPLAN Not.*, 45(10):36–46, Oct. 2010.

[17] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, 1:215, 2010.

[18] S. Jha and S. A. Seshia. A Theory of Formal Synthesis via Inductive Learning. *Acta Inf.*, 54(7):693–726, Nov. 2017.

[19] T. Lau, P. Domingos, and D. S. Weld. Learning Programs from Traces Using Version Space Algebra. In *Proceedings of the 2d International Conference on Knowledge Capture*, K-CAP '03, pages 36–43, New York, NY, USA, 2003. ACM.

[20] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld. Programming by Demonstration Using Version Space Algebra. *Machine Learning*, 53(1):111–156, Oct 2003.

[21] W. Lee, K. Heo, R. Alur, and M. Naik. Accelerating Search-based Program Synthesis Using Learned Probabilistic Models. In *Proceedings of the 39th ACM SIGPLAN Conference*

on Programming Language Design and Implementation, PLDI 2018, pages 436–449, New York, NY, USA, 2018. ACM.

[22] A. Leung, J. Sarracino, and S. Lerner. Interactive Parser Synthesis by Example. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 565–574, New York, NY, USA, 2015. ACM.

[23] P.-M. Osera and S. Zdancewic. Type-and-example-directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 619–630, New York, NY, USA, 2015. ACM.

[24] E. Parisotto, A. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli. Neuro-Symbolic Program Synthesis. *CoRR*, abs/1611.01855, 2016.

[25] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati. Scaling Up Superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 297–310, New York, NY, USA, 2016. ACM.

[26] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In G. Ausiello, M. Dezani-Ciancaglini, and S. R. Della Rocca, editors, *Automata, Languages and Programming*, pages 652–671, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.

[27] N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 522–538, New York, NY, USA, 2016. ACM.

[28] O. Polozov and S. Gulwani. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 107–126, New York, NY, USA, 2015. ACM.

[29] C. S. Ramsey, J. K. Smith, and C. J. Adams. Learning to Learn Programs from Examples: Going Beyond Program Structure. 2017.

[30] O. Roussel. Controlling a Solver Execution: the runsolver Tool. *JSAT*, 7:139–144, 01 2011.

[31] E. Schkufza, R. Sharma, and A. Aiken. Stochastic Superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 305–316, New York, NY, USA, 2013. ACM.

[32] K. Shi, J. Steinhardt, and P. Liang. FrAngel: Component-based Synthesis with Control Structures. *Proc. ACM Program. Lang.*, 3(POPL):73:1–73:29, Jan. 2019.

[33] R. Singh and S. Gulwani. Predicting a Correct Program in Programming by Example. 2015.

[34] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, Berkeley, CA, USA, 2008. AAI3353225.

[35] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster. Path-based Inductive Synthesis for Program Inversion. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 492–503, New York, NY, USA, 2011. ACM.

[36] E. Torlak and R. Bodik. Growing Solver-aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 135–152, New York, NY, USA, 2013. ACM.

[37] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.