Program Synthesis with Constraint Solving 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 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" [21], and it has been portrayed as one of the holy grails of computer science [27, 13]. 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 twofold: we need to find out both how to tackle the intractability of the program space, and how to accurately capture user intent [13].

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.

1.1. OutSystems and 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. To that effect, the kind of programs we are interested in are expressions in the OutSystems language. 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 the user intent.

The most notable success in the area of program synthesis is incorporated in a tool called FlashFill [11]. FlashFill employs PBE technology and is currently integrated in Microsoft Excel. FlashFill is able to synthesize programs that manipulate strings

[1] `https://www.outsystems.com`

very fast, typically under 10 seconds. Another notable PBE synthesizer is SyPet [9], which has been applied in the domain of Java programs.

1.2. 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 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 [9].

2. Terminology and Basic Definitions

This section covers some essential terminology that is used throughout 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: (1) $\mathcal{V}$ is a finite set. It denotes the set of non-terminal symbols. (2) Σ is a finite set. It denotes the set of terminal symbols. (3) R is a finite relation from $\mathcal{V}$ to $(\mathcal{V} \cup \Sigma)^*$, where the asterisk denotes the Kleene star operation. (4) S is a non-terminal symbol. It is the start symbol of the grammar.*

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.

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 [5].

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:

$$\begin{aligned}\phi ::= {} & A \mid p(t_1, \ldots, t_n) \mid t_0 = t_1 \\ & \mid \bot \mid \top \mid \neg\phi \\ & \mid \phi_0 \to \phi_1 \mid \phi_0 \leftrightarrow \phi_1 \mid \phi_0 \wedge \phi_1 \mid \phi_0 \vee \phi_1 \\ & \mid (\exists x.\, \phi_0) \mid (\forall x.\, \phi_0)\end{aligned}$$

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

Definition 6 (Model)**.** *Given a signature Σ, a (Σ-)model $\mathcal{A}$ for Σ is a tuple $(A, (\_)^{\mathcal{A}})$ where A, called the universe of the model, is a non-empty set, and $(\_)^{\mathcal{A}}$ 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: $f(t_1, \ldots, t_n)^{\mathcal{A}}$ is mapped to $f^{\mathcal{A}}(t_1^{\mathcal{A}}, \ldots, t_n^{\mathcal{A}})$; $p(t_1, \ldots, t_n)^{\mathcal{A}}$ is mapped to $p^{\mathcal{A}}(t_1^{\mathcal{A}}, \ldots, t_n^{\mathcal{A}})$; $ite(\phi, t_1, t_2)^{\mathcal{A}}$ is equal to $t_1^{\mathcal{A}}$ if $\phi^{\mathcal{A}}$ is* **true***, and equal to $t_2^{\mathcal{A}}$ otherwise; $\bot^{\mathcal{A}}$ is mapped to* **false***; $\top^{\mathcal{A}}$ is mapped to* **true***; $(t_1 = t_2)$ is mapped to* **true** *if $t_1^{\mathcal{A}}$ is equal to $t_2^{\mathcal{A}}$, and is mapped to* **false** *otherwise. 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.*

3. Background

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 [14], type signatures [19, 10, 22]; syntax-guided methods [2] such as sketches [27], or components [8, 9, 7]; inductive specifications such as input-output examples [10, 12, 18], or demonstrations [17]. 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.

Component-Based Synthesis In component-based synthesis [7, 8, 9, 15] 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 [9], 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 [7].

PBE (Programming by Examples) 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 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 [23]. This can be helpful if it is impractical or impossible to write the output concretely, e.g., if it is infinite.

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 [20, 3] (Section 3), stochastic search [25, 26], and constraint solving [7, 8, 9] (Section 3). Modern synthesizers usually apply a combination of those, enabling us to consider frameworks and techniques for structuring their construction, such as CEGIS [27], CEGIS($\mathcal{T}$) [1], and OGIS [16] (Section 3.1).

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.

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 [7, 8, 9, 27, 15] (typically SAT or SMT solvers). The idea is to encode the specification in a logical constraint whose solution corresponds to the desired program.

3.1. Oracle-Guided Inductive Synthesis

Oracle-guided inductive synthesis (OGIS) is an approach to program synthesis where the synthesizer is split into two components – the *learner* and the *oracle* – which communicate in an iterative *query/response* cycle. The learner implements the search strategy to find the program and is parameterized by some form of semantic and/or syntactic specifications. 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 [16]:

- *Counterexample queries*, where given a candi-

date 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 if p is correct or not. If it is not, the oracle responds with a positive I/O counterexample.
- *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.

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 [27], 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."

The concepts of OGIS was introduced by Jha et al. [15] 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.

4. Synthesis

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. [2]

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.

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 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, and shows input-output examples for each of the functions.

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

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

Figure 1: SSA representation of the program from Example 4.1. The last variable, `r3`, is assumed here to be the return value of the program.

Example 4.1. *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)))
```

4.1. Setwise Encoding

Because OutSystems expressions are composed of self-contained pure functions, this synthesis problem fits nicely in the component-based synthesis paradigm (3). 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.

OutSystems expressions can be represented in 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 4.1 could be written as shown in Listing 1. 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

Function Signatures	Description	Examples
Concat(t: Text, s: Text): Text	Concatenation of the texts t and s.	Concat("", "") = "" Concat("x", "yz") = "xyz"
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.	Index("abcbc", "b", 0) = 1 Index("abcbc", "b", 2) = 3 Index("abcbc", "d", 0) = -1
Length(t: Text): Integer	Returns the number of characters in t.	Length("") = 0 Length("abc") = 3
Substr(t: Text, i: Integer, n: Integer): Text	Returns the substring of t with n characters starting at index i.	Substr("abcdef", 2, 3) = "cde" Substr("abcdef", 2, 100) = "cdef"
+(x: Integer, y: Integer): Integer	Integer addition.	1 + 2 = 3 0 + 1 = 1
-(x: Integer, y: Integer): Integer	Integer subtraction.	1 - 2 = -1 2 - 1 = 1

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

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 synthesizer follows the OGIS model, described in Section 3.1. 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.

Program Formula Let us take the program from Figure 1 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 decoded 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.

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).

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. In general, we can capture these constraints with the following formula:

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

where $M = |I|+|C|$.

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. 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 $l_{r_1} \neq l_{r_2} \neq l_{r_3}$. In the general case:

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

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:

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

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.

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. In general, these properties can be encoded in the following formula:

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

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. In general, this can be encoded in the formula

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

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:

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

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.

4.2. 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. 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.

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 $l_{r_i} = |I| + |C| + i$.

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`.

```
prog(i0, i1):
  c0 = ?
  c1 = ?
  r1 = fa1(p11, p12, p13)
  r2 = fa2(p21, p22, p23)
  r3 = fa2(p31, p32, p33)
```

Figure 2: 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.

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. To encode these properties we need to enforce well-formedness and functional constraints.

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 $|I| + |C| + i$.

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.

For further details we refer the interested reader to Bernardo's Master's thesis [4].

5. Experimental Results

A set of 285,522 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. 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. 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.

We used Z3 version 4.8.5 [6] as the underlying SMT solver. Each benchmark was monitored by the *runsolver* program [24], and restricted to a wall-clock time limit of 600 seconds, and a memory limit of 16 GB. We ran the experiments in an computer with Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz processors.

5.1. Evaluation

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 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 Setwise synthesizer 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 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 a different number of constants. For comparison, we also instantiated SyPet [9], 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. 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 2 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 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 3 shows a comparison between all configurations on three input-output examples.

5.2. Discussion

In Figure 3 we can see that SyPet-All has similar results to W-e3:c1. The large number of components makes the search space intractable. Besides, SyPet 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 2). 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. 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 3).

6. Conclusion and Future Work

In this work, we tackle the problem of synthesizing OutSystems expressions from examples, focusing

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

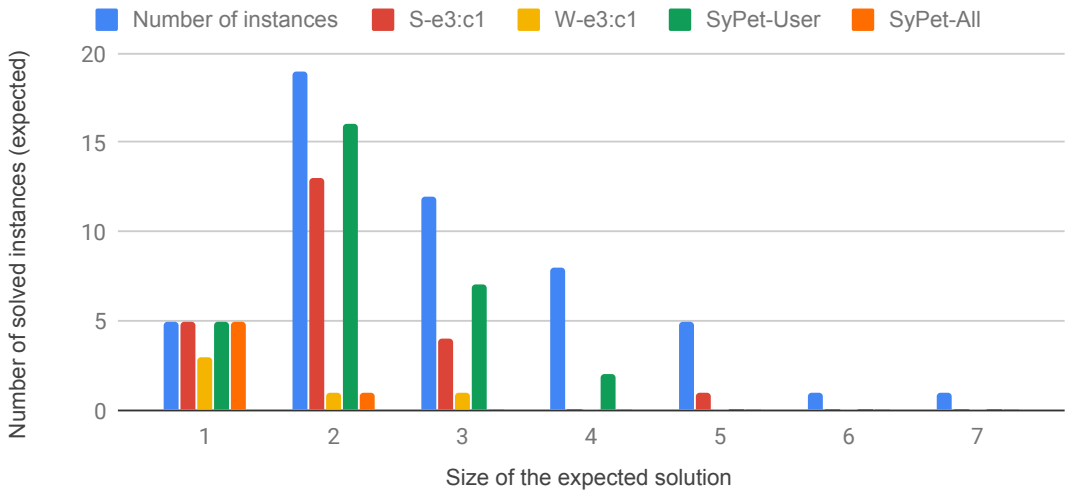

Figure 3: Number of solved instances matching the expected solution per size of the expected solution for the Setwise (one integer constant and one string constant) and Whole synthesizers (one constant) with three examples, and both SyPet configurations (user-provided constants and pool of constants).

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 [9].

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. 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.[3]

Both approaches use encodings whose size scales linearly with the number of input-output examples.

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

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 [7, 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.

References

[1] A. Abate, C. David, P. Kesseli, D. Kroening, and E. Polgreen. Counterexample Guided Inductive Synthesis Modulo Theories. In *CAV*, 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 *FMCAD*, 2013.

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

[4] R. Bernardo. Program Synthesis with Constraint Solving for the OutSystems Language. Master's thesis, 2019.

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

[6] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.

[7] Y. Feng, R. Martins, O. Bastani, and I. Dillig. Program Synthesis Using Conflict-driven Learning. In *PLDI*, 2018.

[8] Y. Feng, R. Martins, J. Van Geffen, I. Dillig, and S. Chaudhuri. Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *PLDI*, 2017.

[9] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps. Component-based Synthesis for Complex APIs. In *POPL*, 2017.

[10] J. Frankle, P.-M. Osera, D. Walker, and S. Zdancewic. Example-directed Synthesis: A Type-theoretic Interpretation. In *POPL*, 2016.

[11] S. Gulwani. Automating String Processing in Spreadsheets Using Input-output Examples. In *POPL*, 2011.

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

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

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

[15] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. *ICSE*, 2010.

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

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

[18] A. Leung, J. Sarracino, and S. Lerner. Interactive Parser Synthesis by Example. In *PLDI*, 2015.

[19] P.-M. Osera and S. Zdancewic. Type-and-example-directed Program Synthesis. In *PLDI*, 2015.

[20] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati. Scaling Up Superoptimization. In *ASPLOS*, 2016.

[21] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Automata, Languages and Programming*, 1989.

[22] N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program Synthesis from Polymorphic Refinement Types. In *PLDI*, 2016.

[23] O. Polozov and S. Gulwani. FlashMeta: A Framework for Inductive Program Synthesis. In *OOPSLA*, 2015.

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

[25] E. Schkufza, R. Sharma, and A. Aiken. Stochastic Superoptimization. In *ASPLOS*, 2013.

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

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