Program Synthesis Co-Design

Managing the tension between interaction models and synthesis techniques

Hila Peleg [1]

[1] *Technion, Haifa, Israel*

Abstract

Program synthesis is the problem of generating a program the user is looking for for them. Since the expressed user intent is often (very) partial, synthesis algorithms must search a space of candidate programs for one that exhibits the desired behavior. A lion's share of the work on program synthesis focuses on new ways to perform the search, but this does not tell the whole story of how a synthesizer operates. This focus disregards the user now, and moreover, makes turning the resulting synthesis algorithm into a usable tool even harder.

The complex relationship between user intent, specifications, and interaction model means program synthesis research must consider all three as early as possible in the development process. Interaction not only dictates the choice of synthesis algorithm but how it must be modified, and algorithmic limitations need to be accommodated in the interaction.

We demonstrate the process of concurrently building both the synthesizer and its intended user-facing tool as a way to search for a balance of the needs of the interaction model (and, implicitly, the user) and the algorithm, a process we name *Synthesis Co-Design*. We include in this paper three case studies of interactive synthesis developed with and without co-design, and discuss the lessons learned from those projects.

1 Introduction

Program Synthesis is the problem of finding a program that satisfies a user's provided intent. Interactive Program Synthesis is the formulation of the problem that includes the user's *interaction* in creating a specification of that intent, processing the result returned from the synthesizer, and refining the specifications if need be. While the first modern program synthesizers were all incorporated in tools [1]–[3], the purely-algorithmic formulation of the problem that is unconcerned with the origins of its specifications has taken the center stage for almost a decade. Recently synthesis research has again returned to tools, becoming more concerned with usability in general and with the *interaction model* surrounding the tool.

The application domain for which a synthesizer is intended affects not only its choice of target language, but also the mode of interaction. For example, FlashFill [1], the earliest successful synthesizer, mimics the interaction of dragging-to-fill in Excel, but instead of inferring a simple range of numbers or progression of dates, or changing the relative references in a duplicated formula, it fills the cells using a computation synthesized using the provided values as examples. Likewise, Rousillon [4] is a synthesizer for web scraping programs with an interaction structured to guide the user into creating specifications describing nested loops to the synthesis algorithm.

Synthesizers for general programming, too, have interaction models, many inspired by developer workflows. Prospector [5] and InSynth [6] are code-completion tools, launching synthesis from type information when the programmer asks for a completion. In CodeHint [3] synthesis is launched from within a programmer's debug session, and additional filters are applied in a "watch expressions" window. In RESL [7] synthesis is part of the REPL's loop. In SnipPy [8] and LooPy [9] inputs that are displayed in the Live Programming environment can be given outputs to form input-output example specifications.

Finding a synthesizer that a proposed interaction model can use can be a challenge: the interaction model dictates the available data in order to form the synthesizer's specification (and, at times, to decide whether synthesis should be launched) and at times adds constraints that should be placed on synthesis results. The available information may not precisely match the information that the interaction model can collect. There might be crucial parts of the specification that the ideal interaction model cannot provide, meaning that the synthesizer cannot be called. The interaction model may

PLATEAU
13th Annual Workshop at the Intersection of PL and HCI

Organizers:
Sarah Chasins, Elena Glassman, and Joshua Sunshine

also have *more* information than the synthesizer accepts, but attempting to constrain results after the fact to match that additional information may cause a search to fail.

Starting from the opposite end is often no better: many synthesis algorithms require specifications that are hard to provide. Constructing large tabular data examples, providing hand-crafted grammars along with the task, or answering an unbounded number of disambiguation questions are all hard enough to construct an interaction around that many projects do not try.

The choice between a synthesizer that cannot run or fails to find programs it should find and a synthesizer that takes hard work to specify is not a particularly useful one. Neither option will make for a usable tool. Our aim, then, is to ensure that we can arrive at the finish line of a project with both a working, correct synthesizer, and a usable interaction. This paper proposes the *synthesis co-design* methodology to this end: an iterative design loop for ensuring the two components converge toward each other.

Specification and search. The design space of a program synthesizer has two main aspects that need to be decided upon: what form of specification will the synthesizer be provided and what search algorithm will be used to find a satisfying program. The synthesizer uses these to search the space of programs that the synthesizer can construct (aka, the search space, the language of the grammar) and return a program if one is found.

While in theory a synthesizer can be generated by selecting any form of specification and any form of search—or, in other words, any element from the *cartesian product* of the two—this does not happen in practice. Why? Because the real design space of synthesizers is a *reduced product* [10]: not every element not every element in the full product describes something that is possible in reality.[1] Since the search space of the synthesizer is astronomical, it must be reduced at least to some degree, to ensure the search can find programs. This is usually done in a way that relies on the type of specifications, making specifications and search tightly coupled.

For example, the search can be enumerative, constructing many programs in the space until one is found. The specifications can then be used to prune the search space on-the-fly by discarding search directions that are nonviable [11], or programs that are equivalent to others seen before [12], [13]. Another approach is to use the specifications construct a representation of the space of programs [1], [14] and search that representation for a solution. While an enumerative search, for instance, *can* be used with any specification, as the exhaustive search will (in theory) eventually cover every program in the space testing each against the specification, this is not a feasible approach. Other techniques simply cannot be instantiated without guidance from a specification.

The tight coupling that leads to this reduced product is well-known in the theoretical side of synthesis research, and helps us explain why, for instance, if a new form of specification is added by the interaction, the synthesizer can now fail to return a correct result: pruning of search directions or constructing a representation of the space based only on some of the specifications may discard the program that will satisfy the new specification. Once new specifications are introduced or the type of specifications has changed, the new point in the design space of synthesizers may not be feasible, and algorithm development will require changing the search in order to find a feasible point.

Interaction and search. Just as synthesis algorithms are a design space, so are *interaction*; Jayagopal et al. [15] present a description of the design space of synthesis interaction models. Together, we can describe an interactive synthesis tool: the joint design space (or product) of an interaction model and a synthesis algorithm. It is probably not a surprising claim that this space, too, is a reduced product: combinations where the interaction model simply cannot provide the synthesis algorithm's required form of specification are easy to think up.

What is perhaps not as intuitive, and is the main claim of this paper, is that it is not possible to select a point in the design space from the outset, using known options for interaction and synthesizer, and build it. Just as specification and search are tightly coupled in making a synthesis algorithm, an

1 This usage of the term comes from Abstract Interpretation where a product of two abstractions is itself an abstraction, but not every element in the product abstraction is feasible. For example, if abstraction A abstracts a number as *odd* or *even*, abstraction B abstracts it as *negative*, *zero*, or *positive*, and we wish to abstract a number using both, then even though $(odd, zero) \in A \times B$, it describes no numbers and we will remove it from the reduced product.

interactive synthesizer is a tight coupling of specification, search, and interaction. This means that, when starting work on a new interactive synthesizer, in all likelihood the point in the design space that describes it does not yet exist.

Honing in on a point in the space can often lead to a mode of work where one of its elements, e.g., the synthesizer, is developed first, leaving the other until later. This means any mismatch between the two has the chance to widen uncontrollably. Working on only the synthesizer puts off the discussion of how its specifications are created, how it is launched, and how its results are consumed. Working only on the interaction ignores changes to the synthesizer that will be needed to accommodate newly information from the interaction, or weaknesses of the synthesizer that the interaction will need to compensate for.

Put another way, though it is customary to think of this point of contact between the interaction and the synthesizer as an interface, with the specification and program passed through it, this is a problematic picture of the situation. One of the purposes of an interface is that it can be agreed upon in advance, then both sides of it can be developed separately, with the other side abstracted away conveniently.[2] But in this case, neither interaction model nor synthesizer can abstract away the other. The entire system is extremely fluid as long as it is in development, and attempting to agree on an interface in advance will ultimately fail.

Synthesis Co-Design. The solution to this conundrum is therefore deceptively simple: interaction design for synthesis must be performed while keeping an eye on the theoretical limitations of the synthesizer, and developing a synthesizer must be done while always considering usability. To make use of new available information in the interaction that is not yet part of an existing synthesizer's specifications, or to collect a new kind of specifications needed by a new algorithm, *both* the interaction model and the synthesis algorithm must be modified—and modified *in tandem*—to avoid the problem of their capabilities diverging.

Co-design, in essence, is about treating the interactive synthesizer as a point in a single design space, rather than two related points in two spaces. It is also the iterative design process of *discovering* that point. Co-design begins, by necessity, with an infeasible point in the space: an interaction idea and a plan to synthesize the specifications collected by the interaction. Continuously working on them in their unified context, even as one or the other is being modified, and constantly striving for a working synthesizer that can be evaluated as a prototype, is the way to ensure the point in the unified space is always a feasible point or at least close to one.

The benefits of co-design. While it may seem like iterating between synthesizer and interaction would be harder work than focusing on one at a time, doing so is not without its benefits. Its most obvious benefit is less work being wasted: a seemingly ready user interface does not need to be modified to start accommodating the synthesizer, and a working synthesizer does not need to be re-developed to adjust for a specification that cannot be collected. But less obvious is the fact that each change in one side can then open up new avenues for the other: changes to the interaction model may make additional specifications available for the search to handle, and theoretical changes to the search can provide more options for the user that can be added to the interaction. This ping-pong between changes made to the interaction and to the synthesizer improves both *while* keeping work from being wasted.

Related design philosophies. The idea of the co-design of the theoretical component and the user-facing one is not new in the PL/HCI sphere: PLIERS [17], a methodology for developing programming languages, includes several phases of iterative design with prototype iteration. Each of these phases involves a theoretical component in the internal iteration that also affect the language design and be affected by prototype changes. Witte et al. [18] outline the importance of maintaining a close contact between a user-facing component and the technical components in long-term projects where incorporating evolving technologies becomes a design consideration.

Others prescribe the relationship between the user interaction and the backend component [19], but an attempt to define an interface between the two components falls into the trap mentioned

2 Though imperfectly, if one believes the Law of Leaky Abstractions [16].

Figure 1. Program synthesis with SnipPy: (a) the user observes values of variables in the Live Programming environment; (b) providing output values creates input-output examples to be passed along to synthesis. Figure originally from [8].

above, generalizing—possibly overgeneralizing—from a single tool.

The structure of this paper. This paper presents the philosophy of *Synthesis Co-Design*, and offers three case studies from the author's own work:

(1) a seemingly perfect marriage between existing interaction and search that still required changes to both,

(2) a failed attempt to match a synthesizer to an interaction model, followed by the delayed implementation of co-design, an d

(3) the expansion of (1), our project most completely adhering to co-design from the start.

Finally, a discussion of the case studies concludes.

2 SnipPy: No such thing as no changes needed

The SnipPy project [8] was a marriage of two paradigms that are seemingly perfect for each other from the start: the Live Programming project Projection Boxes shows variable values at every possible run of the program (Figure 1(a)), and Programming by Example (PBE), a program synthesis paradigm implemented with many different searches, takes input-output examples as its specification. The values in Projection Boxes can be transformed into inputs with user-provided outputs, then sent to a synthesizer to generate an expression satisfying all examples. The initial interaction is very simple: the programmer already sees values for variables from the live programming environment, they can create a new variable, assign output values to some or all of the existing input values (Figure 1(b)), and get a synthesized result for the assignment. Any PBE synthesis algorithm would suffice.

However, even in such a "perfect fit", all was not immediately well. Two problems became apparent once the two components were connected: (1) missing constants and (2) missing examples. Of these, (1) exposes a disconnect between the interaction and synthesizer that was unexpected, and required modifying the synthesizer, and (2) exposed a weakness of the synthesizer that needed the interaction model to compensate for it.

Modifying the synthesizer. The synthesizer we chose for the task was a bottom-up enumerative synthesizer with observational equivalence [12], [13]. It was initially geared toward sygus competition tasks, which meant it accepts a list of constants per-task. While numeric constants can be constructed in the course of enumeration from simpler constants (e.g., `3` is `1+1+1`), the same does not hold for string constants. If the solution requires a string literal such as `";"` it cannot be derived from some basic set of literals. While top-down synthesizers often solve this problem by using a symbolic string literal, then using an SMT solver to solve for the literal, this approach eschews the main benefit of bottom-up enumeration: the ability to synthesize code using components that cannot be encoded for SMT, so long as they can be evaluated.

The synthesizer therefore had to be modified to add *constant discovery* as the first step of the search, inspired by the way Version Space Algebra algorithms handle this problem: if the output includes characters not from the input, they are added to the search's component library. The search can then compose them into any larger needed string literals, the way it uses number literals.

Modifying the interaction. The Live Programming environment displays many different variable values for each line of code. A line can be executed from multiple tests, in a function called multiple times within one test, or simply inside a loop. However, the user may not want to specify outputs for all values. Instead, the interaction model collects as a specification for synthesis only entries for which the programmer provides an output value. Only changed values are passed on to the synthesizer.

It quickly became apparent that this misses out on some important examples: in some cases, the correct output of the example remains unchanged. In the initial interaction, this example was simply not collected, which meant this correct behavior was not reinforced. Because a synthesizer tends to overfit to few examples, this often led to undesirable synthesis results. An addition to the interaction was needed: the programmer could change the output for an input row, or toggle an existing value into becoming an example output.

Added benefits. Even such small changes bring with them added benefits that were not considered originally by the interaction model. Allowing the user to toggle a line to keep its value, for example, brought with it a more natural way to cancel a single example rather than restarting the specification collection: rows could be *un*toggled as easily as they were toggled.

But even more broadly, the interaction of working in assignments to single variables proved to support a mode of incremental, *small-step* thinking about the task, on the interaction side, which led to a synthesizer benefit as well: smaller task slices provided by the user meant smaller goals synthesized in less time, creating a more responsive interactive loop.

3 GIM and RESL: Better late than never

3.1 GIM: Doing it all wrong

The *Granular Interaction Model* [20] was an interaction idea we had in 2016. Synthesis results have, at times, visibly correct and visibly incorrect parts to the solution. Especially in a linear sequence of function calls, some calls or sub-sequences can be outright ruled out and removed from the current space, and other correct parts of the solution might be needed but currently in the wrong place. It would be helpful if, for example, for the program `input.drop(1).take(2).map(x => x.toString())` the programmer could, along with examples, indicate to the synthesizer to exclude candidate programs with the sequence `drop(1).take(2)`, and to include the higher-order function `map(x => x.toString())`.

While the interaction idea was well-liked by developers, and we showed it to also be faster than working purely with examples using a mock synthesizer, we struggled to incorporate a working synthesizer (eventually implemented in [7]).

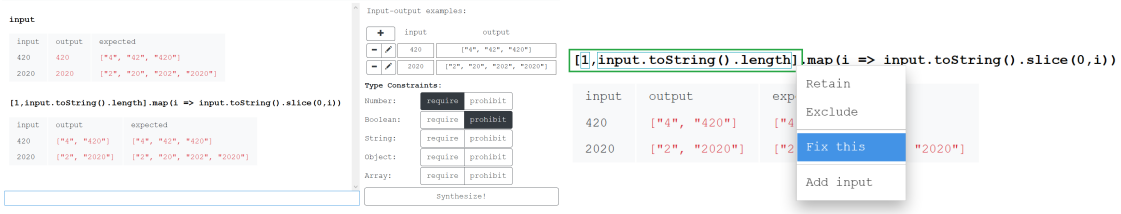
We initially tried to pair it with a bottom-up enumerating synthesizer that searches for longer and longer sequences: `input`, `input.drop(1)`, etc. that search for our desired sequence of operations by length. While operations that constrain the result syntactically trivially prune the search space, and while the originally envisioned interaction was incremental, always pruning a larger portion of the space, this was still insufficient to achieve an even remotely efficient enumeration.

On the other hand, when we started work on an implementation of GIM in a new synthesis tool, RESL [7], the problem was different: if the enumeration is already pruned based on the examples, this can cause the search to be *incomplete*, i.e., miss a program that exists in its search space.

Example. A bottom-up enumerating synthesizer constructs programs from smaller sub-programs that were already enumerated, but when pruning the search using Observational Equivalence [12], [13] uses the evaluation results on examples to divide the space into equivalence classes and, on-the-fly, only keeps one representative from each class and discards the rest.

Assume that the new specification asks to find a program that satisfies the examples

`[3,6,9]` $\rightarrow 6$ `[3,3,3]` $\rightarrow 3$



(a) The RESL Arena, including intermediate type restrictions on the right

(b) Creating a sketch in RESL

.

Figure 2. RESL's user interface. Figures originally from [7].

but also to include the expression `input.length / 2`. Assume also that the enumeration has encountered the subprogram `input[0]`, which evaluates to 3 on both arrays. When we see the subprogram `input.length`, it too evaluates to 3 on both arrays, and we have already seen a representative of its equivalence class, `input[0]`, so `input.length` will be discarded from the enumeration.

With only example specifications this is fine, a program satisfying all examples will still be found. However, if the specification also requires the program to include `input.length / 2`, this is a problem, since we discarded a crucial piece of the puzzle, `input.length`, and will no longer be able to construct `input.length / 2` in our enumeration, so no program satisfying the full specification can be found.

Our synthesizer had to change to accomodate the new specifications.

3.2 Challenge 1: adding new specifications to an existing synthesizer

Once the problem is laid out explicitly, the solution is clear: the new specifications must also be represented in the pruning of the search space. To this end, we generalized Observational Equivalence: the new synthesizer represents each of the specification forms as a family of predicates on program p:

- **Input-output examples:** $\iota \to \omega(p) \triangleq [\![p]\!](\iota) = \omega$
- **Retaining a sub-expression** p': $retain_{p'}(p) \triangleq p'$ is a subtree of p
- **Excluding a sub-expression** p': $exclude_{p'}(p) \triangleq p'$ is *not* a subtree of p

Given the more general domain we selected for RESL, we decided to drop the third syntactic predicate from GIM, *affix*. For each predicate family, we defined a function called an *observer* that returns the value Observational Equivalence will use for given a program. The observer for examples remains the same: $\pi_{\iota \to \omega}(p) = [\![p]\!](\iota)$, computing the value evaluated on the input. For *exclude* and *retain* of a sub-program p', the AST nodes of p' are numbered from 1 to n, and if p is equal to subtree i then $\pi(p) = i$. In any other case, $\pi(p) = 0$.

New possibilities. In order to ensure our new division into equivalence classes will not lead to programs being lost, we set out to prove corretness. This meant reformulating the correctness lemma for Observational Equivalence from [12]. The new, generalized proof showed Observational Equivalence would work with any predicate family, as long as its observer satisfies two properties.

Armed with this information, we could now extend our synthesizer with any new specification that can be observed in a way that preserves these properties. In the paper we hypothesized several such predicates, and chose to add two of them to our interaction model: requiring and prohibiting the use of a type of an intermediate value in the computation, i.e., "the computation should not use strings" or "the computation must use arrays". These are shown in Figure 2a.

In this challenge, a problem the interaction model posed for the synthesizer was resolved, opening up new possibilities for the synthesizer. In the next subsection, we will show the opposite direction, the synthesizer introducing a challenge the interaction must compensate for, but the chosen mode of compensation also requiring a change in the synthesizer.

3.3 Challenge 2: Synthesizing loops

The domain we selected for our new tool RESL was JavaScript programs, where the string and list manipulation supports higher-order functions. These functions, `map`, `filter`, `sort`, and `reduce` are all an abstraction of loops iterating on their list parameter. GIM's initial support of such constructs

included the calls to the higher-order functions as macros, but as RESL was to be a more realistic synthesizer we wanted to synthesize the higher-order function's function parameter.

Synthesizing loops is hard. There are many workarounds, from synthesizing unrolled loops [21] to asking the user for additional examples [12] to incomplete stochastic methods [22] to, at the worst case, collapsing to a full enumeration [11]. We chose a different approach. We decided to support only higher-order functions with independent interations: `map`, `filter`, and `sort`. In such case, *exapmle propagation* [23] is sometimes available as a mode to generate an inner specification to the function parameter, which can then be synthesized separately.

Our idea for a bottom-up synthesizer was similar, though only the inputs are propagated: because of the bottom-up construction, we are not aiming for programs that result in a specific output, but that can be used in building blocks in larger programs. This means that in certain cases where top-down example propagation would lose the information about the output, our synthesizer would work just fine. Unfortunately, this led to a slow synthesizer: it would try all three HOFs for each list value already enumerated, and each of those would yield numerous programs to be added to the space.[3] This weakness was compensated for in the interaction instead, by introducing sketching to the synthesizer and thereby delegating some of the work to the programmer.

Changing the interaction. We had, by this point, decided that the best programmer workflow in which to implement GIM is a REPL (read-eval-print loop), where the iterative nature of editing an existing snippet and re-running it would be suitable for alternating editing and repair-by-synthesis of one-liners (as in Figure 2b). It was possible, then, to decide that the programmer would inroduce a loop themselves, by entering `input.map(x => x)` (i.e., a map that does nothing), then designating the function parameter to be synthesized: `input.map(x => ?)`, and sending it to the synthesizer.

This required adding *sketching* to the synthesizer: adding a sketch to the specification, as well as executing the program up to the hole of the sketch in order to generate the inputs on which synthesis will *actually* run, but on the other hand rather than propagating outputs as well, a process that can lose information anyways, always testing the synthesized completions within the sketch.

Tying up loose ends. Once sketching was added to the interaction model, it could be used outside of loops as well: the user could ask the synthesizer to fix any subexpression, creating a sketch with a hole *anywhere*. Along with the user's ability to write code themselves, including using functions that are ourside the synthesizer's initial *vocabulary*.

It was quite jarring for users to try to fix an almost-working expression that uses, e.g., a call to the function `flat()` to flatten a matrix into a list, only to get back a program that replicates its functionality with other functions. In other words, functions used in the completion of the sketch contain *intent* that was being thrown away.

Fixing this involved another change to the synthesizer: because it enumerates bottom-up, and so evaluates every AST it constructs, it can also evaluate operations it does not know using a language engine. This allowed us to add a phase of *vocabulary enrichment*: at the same time that the variable names that are available in the context are added to the vocabulary, so are any operations or functions from the sketch's old completion. If the user only introduced a loop using `map(x => x)`, `x` is already being added so nothing changes, but in our example, the synthesizer's vocabulary will be updated with `flat()` which the synthesizer will try to use when synthesizing a new completion to the sketch.

4 LooPy: Co-design from the start

One of the major limitations of SnipPy (Section 2) was its handling of loops. Specifically, SnipPy could synthesize programs that perform Python list comprehensions, loops whose iterations are entirely independent, and could be called inside a loop, but could not use the assigned variable in the synthesized expression excluding loops with data dependencies.

Computing an assignment to a variable that also uses that variable using a PBE synthesizer requires the examples to contain the value of the variable before each new assignment, something that cannot

3 The idea was sound, the poor performance was due to shoddy engineering work on this author's part, and synthesis of list comprehensions using this method was implemented in [8] thanks to the superior coding skills of Kasra Ferdowsi.

be refined from end-to-end examples [7] unless those examples essentially encode the intermediate steps, e.g., [11]'s handling of `fold` and [12]'s handling of recursion.

In developing LooPy [9] we saw an opportunity to use SnipPy's interaction model to guide the user in entering consecutive loop iterations that will naturally provide the synthesizer with the needed intermediate steps. We also entered the project armed with a more concrete philosophy of co-design.

Changing the interaction model. SnipPy's interaction model and specifically mode of providing specifications line by line given values of the variables seemed to be a good starting point for getting specifications that provide those internal variable states: if before SnipPy users could select any inputs to provide an output for, creating disconnected examples for synthesis, the interaction model would like to force them to provide examples in order. If they are specifiying the sum over a range, the input interface would look like so:

```
sum = 0
for i in range(1,5):
    sum = ??
```

#	i	sum_{in}	sum_{out}
0	1	0	|
1	2	0	
2	3	0	
3	4	0	

#	i	sum_{in}	sum_{out}
0	1	0	1
1	2	1	|
2	3	0	
3	4	0	

On the left is the sketch into which an expression for `sum` will be synthesized. In the middle is an initial box for entering outputs based on the inputs. In SnipPy, the user could provide an output for any row. However, in order to make sure the examples correctly encode all intermediate values of `sum` for the synthesizer, in LooPy the interaction constrains the user to provide good specifications.

First, the interaction model does not allow skipping iterations in a loop. Once an iteration is skipped, we have no further indication of the value of $\texttt{sum}_{in}$ in any subsequent iterations. In the example in the middle, the user will be barred from switching to any other row. Second, once the user *does* enter an output value in an iteration, this value is used as an oracle to compute the next *in* value for the variable. In this simple example, $\texttt{sum}_{out}$ of iteration i immediately becomes $\texttt{sum}_{in}$ of iteration $i+1$, but if there are additional computations between the current and the next assignment to `sum` they are also performed.

Evolving the entire interaction. Once such programs were working, we were not satisfied: few interesting loop programs have a single straight-line block where we would only want to synthesize one assignment. However, the moment we want to synthesize a *second* assignment, say `x = ??` followed by `y = ??` and the two are interdependent, `x` using `y`, this means the before-state of `y` when used to synthesize `x` was actually incorrect, and we are right back to where we started.

The specifications had to change to handle this case: the user would specify several outputs at once, comprising the effects of the entire block, and the synthesizer would come up with an assignment sequence to support all of them (Figure 3a). This introduces an opportunity to handle tasks where the order of variables to assign is finicky: if the variables specified are unordered, the synthesizer can help the user come up with the assignment order given the values in the examples.

Changing the synthesizer. We now know `x` and `y` must both be assigned synthesized expressions, but we no longer know in what order. Of course, once information was removed from the specification, this impacts the synthesizer, and we must change it to support the new specifications. Specifically, we needed to support any possible sequence of assignments leading to the same final state where all outputs provided by the programmer were assigned.

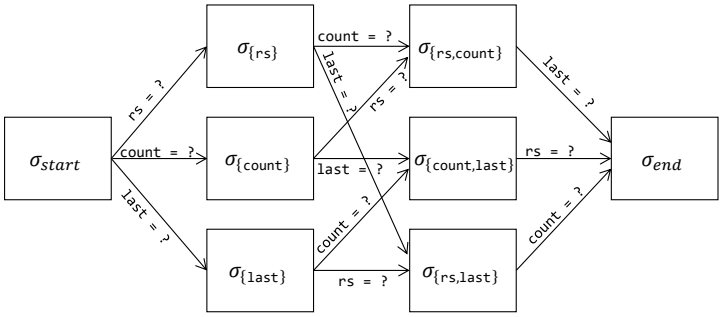
We generalized our synthesis algorithm using the following observation: while k variables have $k!$ possible orderings and any of them could be part of the target program, many of the orderings share intermediate states. For example, when synthesizing `x,y,z = ??`, the initial state is shared between all six orderings, the intermediate state where only `x` has been assigned its provided value is shared between two of them, and the state where both `y` and `z` have already been assigned is also shared between two orderings: assigning `y` and then `z` and assigning `z` and then `y`.

More generally, while there are $k!$ orderings of k variables, there are only 2^k intermediate states including the initial and goal states. When using bottom-up enumeration with Observational Equivalence (which we were), the state before a synthesized expression is key: an enumerator for each

```
1  def compress(s):
2      rs = ''
3      count = 1
4      last = s[0]
5      for c in s[1:]:
6          if last == c:
7              count += 1
8          else:
9              last, count, rs = ??
10     return rs
11
12 compress('aabccca')
```

#	c	s	rs_{in}	$count_{in}$	$last_{in}$	$last_{out}$	$count_{out}$	rs_{out}
0								
1	'b'	'aabccca'	''	2	'a'	'b'	1	'2a'
2	'c'	'aabccca'	'2a'	1	'b'	'c'	1	'2a1b'
3								
4								
5	'a'	'aabccca'	'2a1b'	3	'c'	'a'	1	'2a1b3c'

(a) Entering an output for one iteration in LooPy is used to compute the variable values for the next iteration.



(b) LooPy's intermediate state graph. Each state except σ_{end} holds its own enumerator searching for assignments to variables that will lead to a next state in the graph. This covers $|Vars|!$ orderings in $2^{|Vars|}$ enumerators.

Figure 3. LooPy interface and internal data structure. Figures originally from [9].

single expression is initialized on its input states. Our observation means that intermediate states that are shared can share an enumerator, and the same enumerator can look for expressions that, when assigned, will lead to any of the next possible intermediate states. The LooPy enumerator comprises 2^k OE enumerators in an *intermediate state graph* (Figure 3b), where a solution is a path through the graph specifying the assignments.

Added benefit and one last change. As with RESL, changes to the synthesizer open up new opportunities for the interaction: Once enumeration uses the intermediate state graph, the synthesizer's grammar can be extended to top-level conditionals easily: conditions are searched for at the initial state, and graph edges include which examples they match for, allowing the same data structure to test all possible partitions into *then* and *else* branches (including no conditional) at once.

Once conditionals are added to the langugae, of course, the user can expect to synthesize *inside* of a conditional as well. This meant that the *live execution* that transforms the after-state of a hole into the before-state of the next iteration must support running the computation until the next time a branch is entered, something that did not break any of the components, just needed to be supported.

LooPy was the first project developed after the co-design methodology was formulated. LooPy's development was the smoothest of all three projects, and was even smoother than SnipPy's, despite the fact that SnipPy required far fewer changes to both interface and synthesizer than LooPy.

5 Discussion

Following these examples of implementing synthesis co-design, we conclude by lessons we have learned from working on the interaction and the synthesis algorithm in tandem. We hope other synthesis researchers find these lessons helpful and encourage them to adopt co-design as part of their work.

A more streamlined process. We must begin with the simplest argument for co-design: when working in tandem, less work is wasted. The Granular Interaction Model (Section 3.1) included an *affix* predicate concerned with retaining the initial part of an invocation sequence, which was discarded once we started work on a synthesizer and decided to expand the scope beyond linear sequences of function calls. Moreover, once we decided on a REPL as the base programmer workflow to modify, this introduced the possibility of the user editing code between synthesis iterations, rendering theoretical work on the iterative refinement loop of the original GIM irrelevant.

Design work on the interaction alone caused interaction and synthesizer to diverge, and the work

invested in both could have instead furthered the end product. PLATEAU workshop participants asked for an estimate of the speedup: RESL took almost two years of design and development work, whereas SnipPy took two months and LooPy about four months.

Added benefits. The effort of, essentially, developing two projects at once rather than one at a time *is* a more complex effort with more to juggle. While the wasted effort of trying to develop one and then the other should be a sufficient deterrent, there are also positive outcomes to practicing co-design. Specifically, working on both sides of synthesis in tandem means there is time to recognize opportunities opened up by changes made early on, when acting on them is still convenient and cheap.

The changes we described to the RESL synthesizer enabled us to add any new families of predicates we choose to the synthesizer, which meant we could decide which ones would be a best fit for the interaction we were building. Changing LooPy's interaction made it clear that we had to extend the language to include assignment sequences. And changing LooPy's synthesizer to support sequences made it easier to also support conditionals.

Human solvers. "Human in the loop" notions of program synthesis are often utilized to provide the algorithm with more information. These often take the shape of active learning-style algorithms, turning to the user to disambiguate [24]–[26]. These are the results of centering the algorithm in the design process. However, when the interaction is centered, human in the loop can mean something else as well: how the interaction can guide the user in acting as a solver, providing bits of information that are hard or undecidable for the algorithm to get but given guidance from the interaction easy for the programmer to provide.

SnipPy's small step interaction model encourages the programmer to think in small slices of the target program, thereby dividing up a large synthesis task into multiple smaller ones that are easier to solve. Likewise, RESL asks the user to introduce a sketch by marking a subexpression and indicating "fix this", an action that on the one hand naturally flows from inspecting that same subexpression's intermediate results as the part of the interaction centered on comprehension allows, and on the other hand provides the synthesizer with a smaller sub-problem to solve. LooPy's examples hinge on the affordance of its interface—a table of rows wants to be filled one row after another—and adds a simple constraint disallowing the skiping of rows.

If original human in the loop synthesizers treated the user like a SAT or SMT solver, an entity that can take a complex input and return a complex model that can be considered correct, then shifting our perspective to the interaction allows us to treat the user more like one would treat a deep learning model: a helper that must be delegated to with care and whose answers must be considered in context and whenever possible validated.

End goal. Co-design is an open-ended process, like iterative design is. It is therefore hard to answer questions like how many iterations are needed before convergence. Sometimes (e.g., SnipPy) the starting point is relatively close to a feasible point in the space, which means the process *can* converge quickly. Other times, e.g., RESL, the initial gap that needs bridging before *a* feasible point in the space is reached is larger.

Each time the system is at a steady state, a feasible point in the space, this raises another question: have we converged or do we keep iterating? At this point, the process reduces to classic iterative design: is the current prototype sufficient, or can we incorporate some newly available information or feature to improve it? While it looks like the end goal can shift quite radically in such a design loop, this is not necessarily a bad thing. One can always stop as soon as a feasible point in the space is reached, staying as close as possible to the original goal, but the shift away from the main goal can also be a force for novelty, particularly when it happens thanks to an unexpected ability uncovered in the course of co-design.

Acknowledgements

Many thanks go out to the participants of the PLATEAU workshop at CMU in February of 2023 for their thoughtful and helpful feedback, and particular thanks to Titus Barik, this paper's workshop mentor. And to Dr. Yotam Feldman, for being the best editing money doesn't buy.

References

[1] 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*, ser. POPL '11, Austin, Texas, USA: ACM, 2011, pp. 317–330, ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926423. [Online]. Available: http://doi.acm.org/10.1145/1926385.1926423.

[2] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld, "Learning repetitive text-editing procedures with smartedit," in *Your wish is my command*, Elsevier, 2001, pp. 209–XI.

[3] J. Galenson, P. Reames, R. Bodik, B. Hartmann, and K. Sen, "Codehint: Dynamic and interactive synthesis of code snippets," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 653–663.

[4] S. E. Chasins, M. Mueller, and R. Bodik, "Rousillon: Scraping distributed hierarchical web data," in *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, 2018, pp. 963–975.

[5] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: Helping to navigate the api jungle," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05, Chicago, IL, USA: Association for Computing Machinery, 2005, pp. 48–61, ISBN: 1595930566. DOI: 10.1145/1065010.1065018. [Online]. Available: https://doi.org/10.1145/1065010.1065018.

[6] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac, "Complete completion using types and weights," in *PLDI*, 2013.

[7] H. Peleg, R. Gabai, S. Itzhaky, and E. Yahav, "Programming with a read-eval-synth loop," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, Nov. 2020. DOI: 10.1145/3428227.

[8] K. Ferdowsifard, A. Ordookhanians, H. Peleg, S. Lerner, and N. Polikarpova, "Small-Step Live Programming by Example," in *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '20, event-place: Virtual Event, USA, New York, NY, USA: Association for Computing Machinery, 2020, pp. 614–626, ISBN: 978-1-4503-7514-6. DOI: 10.1145/3379337.3415869. [Online]. Available: https://doi.org/10.1145/3379337.3415869.

[9] K. Ferdowsifard, S. Barke, H. Peleg, S. Lerner, and N. Polikarpova, "Loopy: Interactive program synthesis with control structures," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, Oct. 2021. DOI: 10.1145/3485530. [Online]. Available: https://doi.org/10.1145/3485530.

[10] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," in *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1979, pp. 269–282.

[11] J. K. Feser, S. Chaudhuri, and I. Dillig, "Synthesizing data structure transformations from input-output examples," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 229–239, 2015.

[12] A. Albarghouthi, S. Gulwani, and Z. Kincaid, "Recursive program synthesis," in *International conference on computer aided verification*, Springer, 2013, pp. 934–950.

[13] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur, "Transit: Specifying protocols with concolic snippets," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 287–296, 2013.

[14] T. Lau, P. Domingos, and D. S. Weld, "Learning programs from traces using version space algebra," in *Proceedings of the 2nd international conference on Knowledge capture*, 2003, pp. 36–43.

[15] D. Jayagopal, J. Lubin, and S. E. Chasins, "Exploring the learnability of program synthesizers by novice programmers," in *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, 2022, pp. 1–15.

[16] J. Spolsky, *The law of leaky abstractions*, Nov. 2002. [Online]. Available: https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/.

[17] M. Coblenz, G. Kambhatla, P. Koronkevich, *et al.*, "Pliers: A process that integrates user-centered methods into programming language design," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 28, no. 4, pp. 1–53, 2021.

[18] T. E. Witte, J. Hasbach, J. Schwarz, and V. Nitsch, "Towards iteration by design: An interaction design concept for safety critical systems," in *Adaptive Instructional Systems: Second International Conference, AIS 2020, Held as Part of the 22nd HCI International Conference, HCII 2020, Copenhagen, Denmark, July 19–24, 2020, Proceedings 22*, Springer, 2020, pp. 228–241.

[19] A. Blinn, D. Moon, E. Griffis, and C. Omar, "An integrative human-centered architecture for interactive programming assistants," in *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE, 2022, pp. 1–5.

[20] H. Peleg, S. Shoham, and E. Yahav, "Programming not only by example," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, IEEE, 2018, pp. 1114–1124.

[21] A. Solar-Lezama, "Program sketching," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5, pp. 475–495, 2013.

[22] K. Shi, J. Steinhardt, and P. Liang, "Frangel: Component-based synthesis with control structures," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[23] N. Mulleners, J. Jeuring, and B. Heeren, "Program synthesis using example propagation," in *HATRA*, 2022.

[24] D. Drachsler-Cohen, S. Shoham, and E. Yahav, "Synthesis with abstract examples," in *International Conference on Computer Aided Verification*, Springer, 2017, pp. 254–278.

[25] V. Le, D. Perelman, O. Polozov, M. Raza, A. Udupa, and S. Gulwani, "Interactive program synthesis," *arXiv preprint arXiv:1703.03539*, 2017.

[26] M. Mayer, G. Soares, M. Grechkin, *et al.*, "User interaction models for disambiguation in programming by example," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, 2015, pp. 291–301.