VERSE:
Programming "Programming By Example" By Example

Garrett Tanzer
Harvard College
gtanzer@college.harvard.edu

Alex Wendland
Harvard College
awendland@college.harvard.edu

Jacob Ajit
Harvard College
jacobajit@college.harvard.edu

ABSTRACT

Programming by example is an increasingly popular model for writing programs to automate repetitive tasks, such as those common in data wrangling. However, due to performance limitations, the inductive program synthesizers that enable this interaction model must be specialized to the domain at hand. Frameworks like PROSE [1] have been developed to ease the implementation burden when writing synthesizers for domain-specific languages (DSLs), but they remain relatively inaccessible to all but experts in programming languages and synthesis. We identify three key challenges facing the users of such tools: designing DSLs, writing deduction rules, and debugging. In order to remove these barriers, we present VERSE, a new interaction model for developing inductive program synthesizers *by example*. Rather than write a DSL specification, the developer provides a corpus of programs in a general-purpose language that they want their synthesizer to be able to generate. From these examples, VERSE extracts a DSL and compiles it down to a PROSE synthesizer that is correct by construction. In our user studies, we found that programmers unfamiliar with program synthesis could successfully build synthesizers in VERSE, whereas in PROSE they faced obstacles that proved insurmountable within the time they had. Our proof of concept implementation and experimental materials are publicly available at verse.

Author Keywords

Inductive program synthesis; programming by examples; frameworks; domain-specific languages; deductive inference; search-based synthesis

CCS Concepts

•**Human-centered computing** → **Human computer interaction (HCI);** User studies; •**Software and its engineering** → **Programming by example;** *Domain specific languages; Automatic programming;*

INTRODUCTION

Programming by example is a interaction model for programming in which the user communicates their intent by providing input/output pairs. A program called an inductive synthesizer responds by inferring a program that can compute these mappings, and that will hopefully generalize to the function that the user has in mind. Because this interaction model requires no programming background and can be integrated into GUI applications, it allows the substantial portion of users who lack access to traditional programming to automate parts of their workflows [13]. Even for users who know how to code, specifying intent by example is often faster than writing a program, expanding the class of tasks that can feasibly be automated [2].

Text transformations and data wrangling are the canonical examples of these kinds of tasks; data scientists are often confronted with idiosyncratic unstructured data, which must be parsed and transformed into a more useful format. Microsoft's FlashFill [9] and FlashExtract [10] are technologies that target precisely this use case, helping users transform the content of columns in Excel and extract information from unstructured text files respectively. Note that these domains are extremely specific, compared to programming in the general sense. Existing approaches to program synthesis all have the flavor of enumerative search over the program space; some dispatch synthesis goals to SMT solvers like Z3 [8], but the performance characteristics here are roughly the same: the more expressive and general-purpose the language, the wider and more intractable the program space. It is only by restricting the problem domain and applying specialized optimizations that researchers have been able to build synthesizers that scale to problems of practical interest.

Drawing from lessons learned during the creation of FlashFill and FlashExtract, Polozov & Gulwani generalize the methodology for creating domain-specific inductive synthesizers into a framework called FlashMeta [17], later publicly released as the PROSE SDK [1]. Whereas previously, domain-specific synthesizers would be implemented on an ad hoc basis, PROSE shares the underlying synthesis engine across domains, requiring the developer only to specify a language to sythesize over. They do so by providing a syntax, semantics, ranking function for candidate programs, and *deductive inference rules*, also called *witness functions* and *backpropagation rules*. Conceptually, these rules are just inverse semantics; given an operator's output, they use knowledge of the operator to return its input, or a information about its input if it is not strictly invertible. These domain-specific witness functions are the key contribution that enables FlashMeta's performance, and they ostensibly allow the developer to write a synthesizer without knowledge of the underlying engine.

CHI'20, April 25–30, 2020, Honolulu, HI, USA

ISBN 978-1-4503-6708-0/20/04…$15.00

DOI: https://doi.org/10.1145/3313831.XXXXXXX

However, this claim is questionable. We conclude from a formative user study, our own personal experience, and murmurings in the world at large that PROSE is a difficult tool to use, and exponentially more so for those who do not understand the synthesis engine underneath. We identify three key challenges when interacting with PROSE:

1. DSL design is hard—for correctness, but also especially for performance. Small changes to the DSL design can have drastic performance implications, which depend on the approach of the synthesis engine underneath. In this way, users need to look through PROSE's abstraction barrier to use it effectively.

2. Writing witness functions is cumbersome. Not only are these functions difficult to understand, but manually specifying them is tedious and error-prone.

3. Debugging errors that occur because of, but not directly inside user-written code, is difficult, because no information from the engine's internals is surfaced. While it may be possible for someone who understands synthesis to reverse engineer the process and debug the system as a black box, it is difficult for non-experts to reason about the sources of problems that arise.

In order to ease these tensions, we propose VERSE, a new interaction model attempting to abstract them all away. VERSE lifts the "programming by example" interaction model up a layer of abstraction, applying it to the developer's interaction with the synthesis development framework rather than just the user's interaction with the resulting synthesizer. The developer provides as input a corpus of programs written in VERSE's simple but general-purpose functional language, and the output is a synthesizer that can generate programs within the corpus's domain. By doing so, we have recasted the problem of designing and specifying a DSL optimized for synthesis, which requires programming languages expertise, into the task of writing code, which a capable programmer with domain knowledge should be able to perform. Technically, this interaction model can be achieved in two main steps: first, by casting DSL design itself as a synthesis problem, and second, by collating a library of language primitives with prespecified and composable deductive rules.

VERSE's interaction model solves the aforementioned pain points of PROSE in the following ways:

1. The task of designing a DSL is offloaded to the compiler, meaning that users no longer have to concern themselves with it. These DSLs also improve over time essentially for free as techniques improve, because the domain is specified by the corpus rather than a fixed DSL (though we consider search over the DSL space out of scope for this paper).

2. The user does not need to manually write any witness functions; deduction rules for their programs are composed automatically from primitives in the standard library, whose witness functions were written by synthesis experts. This approach is roughly analogous to automatic differentation, which has improved productivity in machine learning research by pushing derivatives and backpropagation, which previously needed to be programmed manually, into the language runtime.

3. Because witness functions are handled automatically, there is no need to debug this class of errors. The remaining errors are either bugs in the corpus of programs, in which case standard debugging techniques apply, or misalignments between the intended behavior of the synthesizer and the behavior that was inferred from the corpus. The latter of these might be alleviated by probing the user's intent with a mixed-initiative interaction, or by displaying an interpretable representation of the extracted DSL.

We implemented a proof-of-concept version of VERSE as an end-to-end compiler targeting PROSE. In our summative user study using this implementation, we found that programmers could successfully construct a complete toy synthesizer in VERSE in a short period of time, while they failed to implement even a small component of a toy synthesizer in PROSE. Qualitative feedback indicated that VERSE's interaction model is a promising direction for specifying synthesizers, though it remains to be seen whether a fully-fledged version could fully capture common use cases within its general purpose language. Our proof-of-concept implementation of VERSE and experimental materials are publicly available at verse.

RELATED WORK

We highlight relevant prior work, primarily from the body of research on tools for developing program synthesizers. To the best of our knowledge, VERSE is the first work that specifies a synthesizer's domain using a corpus of programs rather than a language specification.

The most directly related line of work to VERSE is, as mentioned in the introduction, Microsoft's series of synthesis products culminating in PROSE. FlashFill, described in Gulwani 2011 [9], and FlashExtract, detailed in Gulwani 2014 [10], are notable for their widespread deployment in Microsoft Excel and Windows PowerShell respectively. They also serve as instantiations of FlashMeta/PROSE [17]'s D_4 framework, which subsumes synthesizers that are *domain-specific*, *deductive*, and *data-driven* (i.e., example-based). VERSE uses PROSE as a compilation target because it is the most mature framework for building synthesizers to date, and it is performant enough to be used in real products.

Alternative frameworks for building domain-specific program synthesizers include Rosette and SyGuS. In Rosette [19], the developer specifies a DSL and its interpreter in Racket, and synthesis is performed by dispatching queries to an SMT solver. SyGuS [5] is an initiative that aims to standardize the input language for synthesis and establish a convention for benchmarking, to facilitate future work in **sy**ntax-**gu**ided **s**ynthesis. SyGuS specifications are closely tied to SMT formulas. There are no immediate barriers to implementing Rosette or the synthesizers competing in the SyGuS competition as an alternative backend for VERSE; in fact, their use of a generic SMT solver instead of deduction rules would enable a faster expansion of the standard library. However, the performance implications of modifications to SMT formulas are difficult

to characterize, so it is unclear whether corpus-based DSL optimizations would be fruitful.

Surprisingly, Osera & Zdancewic [15]'s work on MYTH—a type and example-directed program synthesis tool which can synthesize simple recursive programs in a subset of OCaml—bears a striking superficial resemblance to VERSE. This essentially results from the fact that when one aims to synthesize programs in a general purpose language, there is no DSL to specify, and so the only interface is programming by example. Users of MYTH can even specify helper functions that they think will be relevant, and these will be added to the search space, which approximately corresponds to the presence of primitives in a corpus. However, beyond appearances the approaches are quite different; in VERSE the user and developer are different people, and there is no expectation that the user has programming knowledge, whereas in MYTH the user specifies programs using type annotations. VERSE focuses on domain specialization to produce performant synthesizers for nonrecursive programs, while MYTH succeeds on small recursive programs in a general language but scales poorly to larger instances or instances that don't benefit greatly from type-based pruning.

In the context of compilers, Cheung et al. [7] exploit the benefits of domain specialization in a somewhat generic way, by assembling a collection of domain-specific compilers and picking out components of programs that can be optimized with each of them. Here, this would be analogous to predefining a library of complete DSLs, and choosing among them either at compile time, after the developer has specified he corpus, or at synthesis time, after the user has provided examples. In some sense, this is a coarser grained version of our approach, which breaks down DSLs into primitives and intermixes them based on an implicit specification of the domain, in the form of a program corpus.

PARALLELS

We now draw what we hope is an instructive parallel between the contributions of VERSE and of automatic differentiation in machine learning.

Just a few short years ago, machine learning practitioners manually computed and implemented gradient formulas for each of their models in order to train them. [6] This burden was unnecessary; today, libraries like PyTorch [16] and TensorFlow [3] automatically compute gradients within the language runtime. As a program executes, a computation graph is assembled by composition of language primitives; when the gradient is requested, derivatives are propagated along the constructed computation graph by the chain rule. This has proven a massive productivity and quality-of-life improvement for machine learning researchers, who can focus their time on more experimentation rather than rote derivations. While autodiff is limited in certain fundamental ways, like that it cannot extract useful information from nondifferentiable primitives like control flow, users are happy to accept these limitations. Those who wish to circumvent these shortcomings implement separate algorithms like REINFORCE [20], often in concert with autodiff.

At its core, VERSE also aims to shift unnecessary or redundant labor into the language toolchain. Rather than allow users to define their DSLs or synthesizers in full generality, we restrict their options to a language consisting of a set of primitives and simple rules for combining them. These primitives have special-purpose backpropagation rules in order to achieve better synthesis, much like functions in machine learning libraries define custom, more efficient backpropagation rules, even when those gradients could have been computed by composition of other primitives. We hope that this compromise is sufficient to satisfy the needs of most users, especially when taking into account those who would not have otherwise been able to create synthesizers, and that those users who *do* need to escape the language boundaries still save time by writing the rest of their synthesizer using VERSE's interaction model.

PROSE

We will now return to PROSE and more fully explore its programming model, both in the abstract and in a human-centered context.

Specifying DSLs

In PROSE's interaction model, the developer provides as input a DSL specification, and the output is an efficient synthesizer for that DSL. Such a specification in PROSE comprises 4 components:

1. the syntax, written as a BNF grammar
2. the semantics, written as a collection of C# functions
3. the scoring functions (which are used to rank programs that satisfy all examples), written as C# expressions
4. the witness functions, written in C# but with heavy use of custom-defined attributes

Writing the syntax, semantics, and scoring functions are fairly conventional programming tasks, so they cause few usability issues. Earlier, we identified three key challenges that do exist: designing DSLs, writing witness functions, and debugging. The first of these comes before we actually interact with PROSE: choosing a DSL to specify is a skill that takes experience and many attempts at refinement. In the inaugural presentation of PROSE at OOPSLA 2015, even the creators of PROSE admit that "DSL design = art + lots of iterations" [18].

The second challenge is witness functions. As an illustrative example, imagine we have a function that concatenates two strings, `append(x,y)`. If we know that `append(x,y)="foo"`, what can we say about `x` and `y`? The possible pairings are (`""`,`"foo"`), (`"f"`,`"oo"`), (`"fo"`,`"o"`), and (`"foo"`,`""`). This very basic example is already somewhat more complicated than the forward semantics, which is just `x + y`. Writing these functions in PROSE itself is even harder, because you do not specify properties of all the input arguments simultaneously, as we do above. Instead, you must choose a conditioning order for the arguments, and write witness functions that specify the first argument given the output, the second argument given the first and the output, etc.

The third challenge is debugging, which is closely connected to the previous two challenges. Because the individual components of the DSL are disconnected, all global control flow essentially travels through the synthesis engine, which is intentionally opaque. This means that if your synthesizer runs but does not function correctly, it is difficult to even isolate which component of the DSL specification is incorrect. Despite many hours of effort, we were unable to implement a fully functional pair of witness functions for the append function described above, despite it being an elaborated example in the FlashMeta paper [17]. However, we still do not know whether the witness functions themselves are incorrect, or whether the problem is in interactions with witness functions for other primitives. Even if you annotate all user-provided code with copious print statements, there is not enough information to determine why certain behavior is happening—only where it is happening.

Polozov & Gulwani [17]'s only discussion of usability in their initial presentation of PROSE is a comparison across time to completion and lines of code between the original version of FlashFill, FlashExtract, etc. and new versions reimplemented in PROSE by collaborators. While there is about an 8x factor of improvement in time to completion, this is from 8 months to 1 month, which is still a good amount of time considering that most of the collaborators had prior experience with program synthesis. This comparison is also confounded by the fact that the design of the DSL was fixed during reimplementation, which eliminates much of the time-consuming iteration in DSL design that they lament elsewhere.

Design Study

Though we were initially enticed by PROSE's impressive synthesis capabilities, anecdotes from faculty and others who had used it instilled in us a healthy dose of skepticism about its usability, which our own personal experience eventually confirmed. In order to pin down which aspects were the most troublesome and therefore which to target with our solution, we held a formative study.

Study Structure

The formative study was structured as a cognitive walkthrough. The study was split into two phases, an educational phase and then a challenge phase. A study coordinator recorded notes throughout both phases because, even though our focus was on framework usage and not learnability, they are intermingled concepts that can be hard to pry apart.

Furthermore, since we aim to make synthesis accessible to programmers at large, we selected two CS-adept college seniors with software engineering experience but no formal exposure to program synthesis. We requested that they verbalize their thinking throughout a structured in-depth interview:

1. (Educational) Program Synthesis Introduction and Demonstration
   (a) Subjects were given a writeup[1] with an introductory paragraph about program synthesis and the PROSE framework. A second paragraph described the demonstrations they were about to see, with an overview of the primitive operators that had been implemented in the framework: Substring and two accompanying absolute (AbsPos) and regex-based positioning (RelPos) operators. Importantly, the absolute position operator did not have support for negative indices, so it could not handle the extraction of data at the end of variable length strings.
   (b) Subjects were then presented with an interactive REPL allowing them to interact with a PROSE-generated synthesizer. They were given input/output examples to feed to the synthesizer and test the resulting program.
   (c) Subjects were shown a video demonstration of the FlashFill program synthesis capabilities of Excel (see the Summative Study for video details). They were asked to describe the relationship between the program synthesis in the video and in the REPL. If their answers were incorrect or unclear, they were given a correct description by the study coordinator.
   (d) Subjects were presented with a second interactive prompt and corresponding input/output examples to synthesize a new program.
2. (Challenge) PROSE Source Code Review
   (a) Subjects were shown the source code used to implement the synthesizer they had been interacting with.[2] They were shown the grammar, semantics, ranking score, and witness function files that PROSE needs to operate.
   (b) They were asked what the purpose of each file was, and were permitted to review online PROSE documentation as desired, but were not corrected if their understanding was wrong.
3. (Challenge) New Synthesizer Feature
   (a) Subjects were shown a new set of input/output examples that required operators which could extract substrings at the end of variable length strings. They were informed that adding support for negative indices in the AbsPos operator would enable this. They were asked to add this support to the PROSE configuration they had been reviewing previously.

Findings

Each subject spent around one hour attempting the challenge. Neither was able to complete it. Both recognized that no changes were needed to the grammar or ranking score files but changes were needed for the semantics and witness functions. Both successfully updated the semantics to support negative indices. Both identified the correct witness function to update, though neither recognized which update should be made. After prompting them with instructions taken from an online PROSE tutorial,[3] both identified where the change needed to be made but were still unable to identify what the change needed to be.

[1] See accompanying materials for examples of formative study instructions that were provided to subjects.

[2] The source code is hosted at prose-formative.

[3] `ProseTutorial/synthesis/WitnessFunctions.cs` from the DslAuthoringTutorial Part 1c on GitHub.

VERSE

In contrast, VERSE replaces the input of DSL specification with a *corpus* of programs written in a general-purpose purely functional language. From this corpus, a compiler infers a domain specific language, which is constructed in such a way that we are given its syntax, semantics, and witness functions without requiring user intervention. By default, we generate scoring functions that rank programs by length. This interaction model is depicted graphically in Figure 1. By eliminating the explicit user-facing DSL representation, we preclude our 3 key usability challenges entirely. Domain experts who want to create a synthesizer can now apply their knowledge in a more straightforward fashion, by picking and writing code for some representative programs from the domain in a toy language. Furthermore, these representative problems likely have already been collected in a centralized fashion: for instance, FlashFill [9] was validated against a benchmark of 100 representative examples that the Excel team had gathered of their own accord.

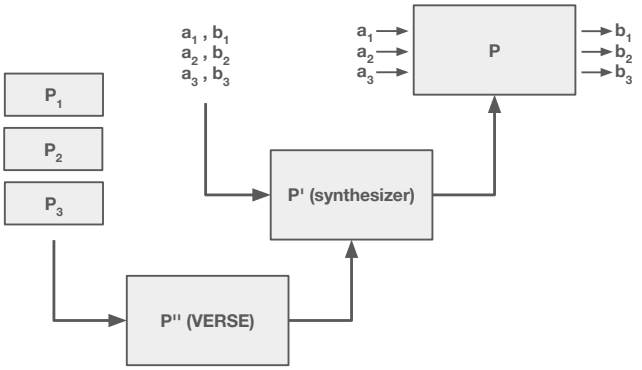

Figure 1. VERSE's interaction model.

Two User Stories

Consider two users working on very outwardly different pieces of software:

1. Jaime is a programmer at FileOverload Co. FileOverload Co sells software for managing large numbers of files and is popular with design studios, photographers, and invoicing companies. It helps users with organizing large sets of files into tagged categories for easy organization and discovery later. Many users find this tagging helpful, but Jaime finds it really inefficient that they have to manually tag files (even though they can select multiple and tag as a batch). Jaime thinks it could be automated, but doesn't want users to have to write out rules for it.

2. Margaret is a programmer working at GroceryPlus!. GroceryPlus! sells an enterprise resource planning (ERP) system for small grocers to manage their inventories. GroceryPlus! has decided to start adding bulk workflow capabilities to their ERP (e.g. renaming inventory and increasing the sum totals if it matches certain criteria). Margaret is tasked with designing the UI for these operations and implementing them. The ERP's interface is already super complex, and if it had dropdowns to support all the operations it could quickly explode into dozens and dozens of permutations. Margaret doesn't think that their end users will be able to learn and remember all of these options for the complex workflows that they need.

Jaime and Margaret are both capable programmers who are focused on bringing meaningful features to their users as swiftly as possible. They've looked into program synthesis before and were particularly impressed by the FlashFill capability in Excel (how does it learn what they want after only an example or two!?). They know their users could show examples of what they wanted or demonstrate the chain of actions in their workflows, but how would they start in implementing that? They don't have an intuition for how program synthesis is implemented, and when they look at the many configuration requirements in PROSE they struggle to see how those details map to their feature goals. It feels intractable...

Until they run across VERSE. They run the demo and it performs close to the same as PROSE, but when they look at the configuration all they see is a small set of example programs that were provided. Uncertain what more it can do, they read the documentation and see a general purpose functional language with a wide range of operators. They try expanding the demo with an append operation and are happy to see that their new synthesizer can support a whole new range of examples! Looking at the input corpus again they start to see how their automated features could be represented as these simple programs. They were going to have to write programs anyways, and though they don't use purely functional languages all the time, they understand what's going on. In any case, they feel quite confident that they can iterate to what they need, and they now see the problem in two distinct parts: 1) how to represent the user activity as input/output examples, and 2) how to represent the workflows as chains of these operators. Jaime knows he'll represent the files as before vs. after records/objects. He expects he can represent the workflows as data extraction on these records, filter operations, and then insertions of new metadata. The configuration won't write itself, but Jaime (and Margaret) are confident it's tractable.

IMPLEMENTATION

We implement a proof-of-concept version of VERSE in OCaml, as a compiler from a toy functional language down to PROSE. The compiler has a lexer/parser (generated using Menhir), typechecker, DSL extractor, DSL optimizer, and backend that generates a PROSE project. Currently, the optimizer is instantiated as a no-op.

Input to the compiler comes in the form of a text file `corpus.txt`, which contains input/output type annotations and input name:

```
@input type id
@output type
```

as well as a list of programs delineated by semicolons. The toy language of these programs supports functional application, let constructs, and not much else. We omit a description of the language's syntax and typing rules because they are uninteresting.

The DSL extraction step outputs an intermediate representation for DSLs that is designed to ease compilation to BNF. On the first translation pass, all primitives in the source language are transformed into a flat representation (i.e. one that does not store arguments for functions). This represents a simple DSL optimization where primitives that are not present in the corpus should be culled from the grammar. In subsequent passes, sequences of operators in the original program can be nested within inductive constructors in order to represent that those primitives should be composed into a single one (though we currently do not do so). Given this definition of an operator, we define a DSL as a 3-tuple of an input type, output type, and map from types to sets of operators.

Most of the backend that compiles to PROSE is trivial. For semantics, ranking functions, and witness functions, we can think of source files as simply a set of functions; if a primitive is in our DSL, its library functions should be in the correct source file. We store our standard library as a collection of text files. Compiling the DSL to a BNF is slightly more involved, but simplified greatly due to our DSL IR. Each type that keys into the map corresponds to its own BNF rule; the set of operators that is the value in the map for that rule corresponds to the rule's constructors. For operators that have nested constructors, special singleton grammar rules are created to reconstruct the term in the BNF. We express the composition in the BNF rather than giving PROSE a flattened representation, because this way PROSE will compose witness functions for us as part of its normal deduction procedure. As the nonterminal rules are singletons, this preserves the same number of states visited as if we had composed the witness functions above the level of PROSE.

Limitations

The main limitation of our VERSE implementation is its tiny standard library of 3 operators. The difficulty here was not with integrating a standard library into the compiler—the architecture is supports swift addition of new functions—but rather with defining these primitives in PROSE itself. In some sense, this underscores the need for improved interaction models in the vein of VERSE.

The main frustration we experienced when trying to implement primitives, beyond the debugging experience we discuss in Specifying DSLs above, was PROSE's inadequate documentation. For example, PROSE provides several "standard concepts" for enumerable data structures, which are supposed to provide witness functions out-of-the-box. However, when we tried to instantiate an operator with the `Kth` concept below, which indexes into an enumerable,

```
Kth(seq: IEnumerable<T>, k: int): T
```

deduction failed silently (i.e., the synthesizer could not generate any programs containing `Kth`). After prolonged debugging, our best guess is that `Kth` actually only supports one witness function, for the parameter `k`, and assumes that `seq` is the input token. When we tested the synthesizer with a grammar that satisfied these conditions, `Kth` started appearing in generated programs. This requirement is not indicated anywhere in the (scant) documentation for `Kth`.

To be broadly useful, VERSE would need to expand its standard library to encompass a wide range of operators. We discuss this more in the Summative Study.

Summative Study

In order to evaluate the success of our new interaction model, we conducted a user study with 5 new participants, none of whom had any formal exposure to program synthesis. We again selected participants who could be the target audience for a program synthesis framework: they were all 4th-year undergraduates concentrating in computer science who had plans to work in software engineering industry or academia.

We wanted to test meaningful differences between existing methods (i.e. PROSE) and our approach (VERSE). Therefore, our preferred method would have been a within-subject design where each participant used both methods. However, as we learned through the design study, familiarizing oneself with PROSE is a taxing process, and we didn't want to encumber our participants with more than 1.5 hours of tasks.

Subsequently, we had intended to do an A/B test. However, upon reflection we realized that we expected VERSE to be clearly superior at the tasks and therefore an A/B study would not extract the most useful information. Instead, we structured the study as a think-aloud usability evaluation, to provide richer qualitative data around the pain points that VERSE still has.

With the think-aloud usability evaluation, we decided to have most participants use VERSE instead of splitting between VERSE and PROSE evenly since we assumed the results concerning PROSE from the design study would persist. To sanity check this assumption, we had one user attempt the challenge using PROSE, permitting access to the online PROSE tutorial, and confirmed that they exhibited the same difficulties.

Study Design

Since we saw our target users as those who may benefit from program synthesis techniques but a) were currently not familiar with them and b) may be dismayed by the complexity of current approaches even for simple operations, we constructed a challenge to implement a subset of the features in the PROSE tutorial.[4] The study was composed of an educational phase followed by an assessment phase. Subjects were asked to communicate their current thinking throughout the study, and informed that they would be prompted to share their thoughts if needed.

A study coordinator was present to record the subject's commentary, to answer questions, and to provide Wizard-of-Oz services as necessary (2 subjects used features that needed Wizard-of-Oz handling). Study coordinators conveyed beforehand that they would be unable to answer questions related to the VERSE or PROSE framework or their accompanying documentation, but that they could assist with clarifying the challenge presented to the user in the writeup. Furthermore, the study coordinator provided C# syntax assistance for subjects using PROSE.

[4] https://microsoft.github.io/prose/documentation/prose/tutorial/

Figure 2. A mockup of *Data Cleaner 3000*, a hypothetical application that helps end users extract relevant structured data from semi-structured data sets by applying transformations to each row that have been learned through a small set of end user provided examples.

Potential Program	Input Example	Output Example
substring(x, -5, -1)	"Amanda was born on February 8, 1997"	"1997"
	"Rachel was born on June 6, 2003"	"2003"
substring(x, 2, -2)	"(Amanda Smith)"	"Amanda Smith"
	"(Oprah Winfrey)"	"Oprah Winfrey"
substring(x, 3, -3)	"(~Jacob Patterson~)"	"Jacob Patterson"
	"(~Maxwell Martin~)"	"Maxwell Martin"

Table 1. The set of three groups of input/output examples provided to the subject as test cases that their synthesizer needed to support. Note that *Potential Programs* were not shown to the user, instead, they are included here purely as representation of what a successful synthesizer may have produced.

The study was broken down as follows:

1. (Educational) Program Synthesis Introduction

   (a) Subjects were given a writeup[5] explaining program synthesis and showcasing the FlashFill program synthesis capabilities of Excel[6]. Then, the writeup informed them that they were on a team building a data cleaning application (see Figure 2 for a mockup of what they were shown). They were tasked with implementing the program synthesizer that would be the backbone of the data cleaning application. They were given a set of input/output examples they needed to support (see Table 1 for the list). Finally, subjects were asked a set of check-in questions to ensure they understood the introductions and task at hand. The study coordinator would prompt them to review parts of the writeup or FlashFill demonstration video if the subject demonstrated confusion.

      i. What is program synthesis?

      ii. What was the challenge you were tasked with building a solution for today?

2. (Educational) PROSE/VERSE Framework Description

   (a) The writeup concluded with an overview of either PROSE or VERSE. Both frameworks were introduced as tools out of Microsoft Research that we were assessing the usability of. These overviews described the project templates the subject would start out with, how the synthesizer's REPL worked, and where to find additional documentation[7].

3. (Assessment) Challenge

   (a) Subjects were provided a project template base with either VERSE or PROSE installed and a base set of necessary files provided for each (a corpus for VERSE,

[5] See accompanying materials for examples of summative study instructions that were provided to subjects for both VERSE and PROSE.

[6] A YouTube clip from the FlashFill research team in 2017 was shown, from approximately 0:18 to 1:40.

[7] For PROSE, the main project site was provided.

and {Semantics, RankingScore, Grammar, Witness Functions} for PROSE). Subjects were given a terminal in the directory, a Visual Studio Code IDE (which all subjects expressed familiarity with) open to the project, and a file browser in the directory.

(b) For VERSE, since there wasn't an online site like PROSE had, subjects were also provided a documentation file[8] in the directory. This file contained an overview of the functional programming language that VERSE supported as well as a list of 21 operators that were supported in this programming language, such as `join(l: string[], d: string) -> string` and `append(s1: string, s2: string) -> string`. The implementation of VERSE that was being used during the study only supported a subset of these operators, so if a subject chose to use one outside of the implemented set, the study coordinator would Wizard-of-Oz act as the REPL and synthesize programs or report errors as appropriate. 2 subjects used unimplemented operators, while the 2 other VERSE subjects used only implemented operators and therefore did not require Wizard-of-Oz activity.

4. (Assessment) Closing Questions

(a) Subjects were asked a set of purely qualitative questions to see if/how their understanding had changed throughout the session.

i. What is program synthesis?

ii. What was the challenge you were tasked with building a solution for today?

(b) Subjects were also asked a set of mixed qualitative/quantitative questions, with an instructions to provide a response on a 1 to 5 scale, with a preference that they do not pick 3 unless they feel overwhelmingly compelled to.

i. How easy was it to work with the tools?

ii. How good was the tool's documentation?

iii. How good was the system design of the tool for the task of defining a program synthesizer?

iv. How powerful do you think the tool is?

v. How likely would you be to implement a feature using program synthesis?

vi. How likely would be to use this tool, taking it as a given that you are going to use program synthesis? As in, would you spend time looking for another tool (5 - no, 1 - yes)?

Findings

3 of the 4 subjects using VERSE completed the task. The VERSE subject that did not complete the task (Subject S5) also had the most confusion about what program synthesis was after the Program Synthesis Introduction portion of the study. Conceptually, Subject S5 described a loose version of program synthesis, *"You give a few examples and then some algorithm generalized those examples into a set of rules for inputs and outputs."*, but upon being questioned about where in the FlashFill demo certain rows were inputs to the synthesizer and others inputs to the synthesized program, Subject S5 was unable to correctly identify them. In addition, Subject S4 appeared to confound the synthesizer and its output programs as well, stating:

> I feel like it was kind of a weird mixture between programming and not programming, and if I was going to define the corpus, I needed to think about it programmatically which is no different than how I normally do. It would have been easier to just write the programs in Python.

This confusion aligns with a general understanding that, like all metaprogramming, **writing program synthesizers is unintuitive**. This may be exacerbated by programming by example paradigm used in VERSE. PROSE has a clear delineation between the synthesized programs and the configuration for the synthesizer, since the synthesizer configuration is written in a different language, C#, in a different environment, in multiple C# project files, compared to a short output program in a purely functional language. Comparatively, VERSE is configured using programs that are written in the same language as the synthesized programs and is configured with short snippets that are barely longer than the synthesized programs themselves. Therefore, since program synthesis is a technique that is likely unfamiliar for most people, a program synthesis framework like VERSE would need to more cleanly delineate the role it serves. A simple first step may be to separate the build and run steps for the synthesizer. (The current build script performs them automatically in sequence.)

Including the Design Study, the 3 PROSE subjects were unable to complete their respective tasks. Additionally, none of their witness functions, as assessed by the study coordinator, were conceptually correct at session conclusion. The PROSE subject (Subject S2) in the Summative Study, however, expressed an appropriate conceptual understanding when asked what the witness function needed to do (Subject S2 was implementing a substring operator that could return the first n-characters): *"So I can take the length of the output and then return the length of the string as the spec."* However, Subject S2 was unable to implement this in a manner that complied with the witness function API and was unconfident about their approach, seemingly due to their inability to create a runnable configuration. Given that PROSE subjects were unable to complete a small modification of an existing project, and VERSE subjects succeeding in replicating the functionality of the entire project, we believe that **synthesis frameworks can be made easier**, at least for common tasks. This admittedly needs to be tested on a more fully-fledged version of VERSE for more meaningful results.

Additionally, most VERSE subjects expressed concern about the power of the framework. Subject S3 stated:

> There's a lot of abstraction and I know there's supposed to be a lot of abstraction, but the design of the system didn't

[8] See DOCS.md in the accompanying materials for a full example of this file.

ID	Track	Succeeded	Q4(b)i	Q4(b)ii	Q4(b)iii	Q4(b)iv	Q4(b)v	Q4(b)vi
Subject S1	VERSE	Yes	3	-	-	-	3	4
Subject S2	PROSE	No	2	3	4	4	2	2
Subject S3	VERSE	Yes	4	2	2	2	2	3
Subject S4	VERSE	Yes	4	4	2	2	1	5
Subject S5	VERSE	No	4	5	4	1	1	2

Table 2. Questions are labeled according to their entry in the Summative Study Design (Section 6.2.1). The qualitative commentary provides more insight, but in general aligns with these quantitative results. Two difficulties seen here are the lack of interest in using program synthesis in the future, and the belief that the VERSE framework is not very powerful.

> help me understand what was going on. If something went wrong I don't know how I would fix it. [...] I don't really know how powerful it is. It didn't just scream power. Though I guess it did work, eventually.

We interpreted this in several ways.

First, we thought there might be precedents of program synthesis-like tools suffering from a Gulf of Execution. As defined in *The Design of Everyday Things* [14], the Gulf of Execution is the gap between what a user wants to do with a system and what the system can actually do. Because of this precedent, subjects aren't giving the system the benefit of the doubt, and instead are requiring it to prove to them that it is highly capable. Our challenge didn't help with that, as conveyed by another of Subject S3's comments:

> [VERSE] worked 100% for the task, but that task didn't feel like it would generalize hugely.

Second, this may be a limitation of the implementation of VERSE currently (since the PROSE subjects found PROSE more powerful). VERSE may be presenting too simplistic of an interface, hiding away too much of what's going on. Selectively exposing these underpinnings without overwhelming the user may be a valuable addition; in the case of VERSE, even showing the generated PROSE configuration may ease this concern and allow users to understand that the functionality underneath is the same.

Third, the 21 operators that were shown to the subjects as being implemented in VERSE are certainly fewer than would be needed for practical use. For comparison, the Python standard library has over 7,000 built in methods,[9] the Node.js standard library has over 1,700 methods,[10] and Java has over 14,000 methods[11]. Though 21 seems like a large number in the context of authoring interoperable primitives in PROSE within a limited time window, in hindsight 21 operators is grossly smaller than any practical language, even if we exclude libraries that would be inappropriate in a purely functional synthesis tool. This is reinforced further by additional commentary from Subject S3:

> I'd prefer to start with a huge library of pre-built corpus stuff. Like all of FlashFill. So I could just build off of that. But maybe there are just applications that I'm not thinking of.

[9] https://devdocs.io/python 3.7/
[10] https://devdocs.io/node/
[11] https://devdocs.io/openjdk 8/

Based on this, we believe that **program synthesis historically suffers from the Gulf of Execution**, similar to the challenges experienced with audio assistants [12], so it's important to clearly expose powerful functionality to users without hiding too much detail. Additionally, there are still limitations in the synthesis framework, so we believe that **program synthesis needs to be transparent**, so that developers may understand what it's capable of. Having a more expansive standard library would address the first of these concerns. To address the second, VERSE could allow users to break the abstraction barrier if need be, and write lower-level code (i.e. witness functions) for new primitives.

Shortcomings

The summative study provided valuable insight, but it leaves certain important questions unanswered.

1. **Can VERSE be used for real-world/complex domains?** This question is difficult to answer without undertaking significant implementation effort. A future study might take an exploratory approach: instead of asking a subject to implement a synthesizer for a set of examples, the subject could be introduced to program synthesis and asked to brainstorm desirable applications. They would then be given a tutorial of VERSE and granted free rein to try implementing their aforementioned applications. This would help avoid introducing the explicit and implicit biases of the VERSE authors about what its capabilities and constraints are.

2. **Can VERSE interplay at multiple abstraction levels?** The study tested the ability to use the framework to produce simple synthesizers for substring-based programs. It's unclear if VERSE can effectively grow into a role where a user must expand its standard library with custom operators. A future study could provide users with a pre-existing VERSE configuration, train them on writing custom operators, and then ask them to author and integrate a new custom operator. Technically this should not be difficult, but designing a clean way to work at two levels of abstraction simultaneously can be challenging.

FUTURE WORK

Beyond the design questions raised above, in particular about scalability and transparency about limitations, there are two main directions for future work: improving overall tool maturity and exploring DSL synthesis.

Tool Maturity

There are a number of additional quality-of-life features that would improve VERSE's usefulness. Most of these are not particularly novel, though some would make it easier to perform a more conclusive user study.

1. The most salient of these is a larger library of primitive functions. Having access to the equivalent of a standard list and string library would make it easy to design a relatively complex user study, and would hopefully convince users that they are not losing any appreciable power by working at a higher level of abstraction.
2. Currently, a DSL's compiled syntax has a nonterminal rule for each C# type that some non-nested expression evaluates to. However, there are some PROSE primitives that we want to represent with basic C# types, even though they are conceptually incomparable. Adding named types to our general purpose language would accomplish this.
3. There are a variety of richer language constructs that can be inverted with witness functions: n-tuples, record types, match expressions, and even lambda functions (in some cases) could be added natively.
4. The user should be able to test the programs in their corpus on various inputs using an interpreter.

DSL Synthesis

A potential direction for future work with more research content is to explore the space of DSL synthesis. That is to say, given a corpus of programs written in a general-purpose language that represent a domain, how can we find a domain-specific language such that synthesis cost is minimized? If we can outmatch or even come close to human performance for DSL design, this is a strong argument for VERSE's interaction model. Here, a domain-specific language is defined as a set of primitives formed by composing primitives in the general-purpose language. This means that the set of valid expressions in the DSL is a subset of those in the GSL, reducing search space.

Now we can cast this as an optimization problem; as a proxy for expected synthesis cost, which we want to minimize over our domain, we will instead use empirical synthesis cost over our corpus. Once the problem is formulated in this way, there are two main contributions to make: first, to define a cost estimate (or, in machine learning terminology, a loss function) over the search space, and second, to actually develop an algorithm for search. This two-step pattern is a common approach for research automating tasks that were previously considered to require great human insight [4, 11].

We have some preliminary ideas for how to accomplish these goals. First, we describe an upper bound on synthesis cost in engines like PROSE that can be computed much more efficiently than actually using the synthesizer. In preparation, we annotate witness functions with a term that upper bounds their fanout. If the synthesis engine traverses the state space in a breadth-first pattern, counting the length of the program by the number of syntax rules with more than one constructor (i.e. those that require us to add a state to the queue, rather than just following the single allowable transition), then the following algorithm computes an upper bound on the number of states appended to the queue to synthesize a program of length ℓ:

1. Create a counter for each rule in the DSL's grammar. Initialize them all to 0 except the one corresponding to the output, which is initialized to 1.
2. Loop ℓ times.
   (a) Create a new set of counters, all zero-initialized.
   (b) For each r_i corresponding to an old nonzero counter :
      i. If there is only one constructor, follow it and repeat.
      ii. For each constructor c_j for r_i :
         A. For each witness function w_k for c_j :
            Multiply the current counter value by the upper bound on w_k's fanout, and add it to the counter corresponding to w_k's argument.
   (c) Replace the old set of counters with the new set of counters.

If r is the number of nonterminal rules in the syntax, c is the maximum number of constructors in a rule, w is the maximum number of witness functions per operator, n is the maximum fanout for a witness function, and ℓ is the length of the program to be synthesized, the time to compute this cost upper bound is proportional to $\ell rcwn$. This follows from the control flow, with the exception of the decision to follow rules with only one constructor. The cost of performing actual synthesis scales exponentially in ℓ. This cost estimate could be validated by taking random samples from the program space and computing the empirical synthesis cost for the functions that they specify.

Second, we describe a first pass approach for optimizing DSLs. Given the cost estimate above, which can be efficiently computed, we can perform enumerative search over the space of DSLs that feature as primitives only those n-grams of primitives in the general purpose language that are extant in the corpus. If an n-gram is not present in the corpus, any DSL including it is strictly suboptimal, which we can use to substantially prune the search space. Because DSL synthesis only happens once (until the corpus is changed) and is massively parallelizable, the cost of traversing this space is not a major concern.

CONCLUSION

Making program synthesis more accessible could enable fundamentally new applications and workflows. In this paper, we presented VERSE, an interaction model for developing inductive synthesizers that can be described as programming "programming by example" by example. In this way, the esoteric task of creating a synthesizer is reduced to the more accessible task of writing a few small functional programs. Our key contributions are to identify three main barriers to entry for existing tools like PROSE—designing DSLs, writing witness functions, and debugging synthesis engines—and to propose a new abstraction that both obviates these obstacles and can be practically realized, at least as a proof-of-concept

implementation. Our user studies provide further evidence that our approach has merit, and there are many avenues for future expansions along design, technical, and evaluation axes.

REFERENCES

[1] Accessed December 2019. Microsoft PROSE SDK. https://microsoft.github.io/prose/. (Accessed December 2019).

[2] Accessed December 2019. xkcd: Is It Worth the Time? https://xkcd.com/1205/. (Accessed December 2019).

[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR* abs/1603.04467 (2016). http://arxiv.org/abs/1603.04467

[4] Omid Alipourfard, Jiaqi Gao, Jeremie Koenig, Chris Harshaw, Amin Vahdat, and Minlan Yu. 2019. Risk Based Planning of Network Changes in Evolving Data Centers. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. ACM, New York, NY, USA, 414–429. DOI: http://dx.doi.org/10.1145/3341301.3359664

[5] 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. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. 1–8. DOI: http://dx.doi.org/10.1109/FMCAD.2013.6679385

[6] Atilim Gunes Baydin, Barak A. Pearlmutter, and Alexey Andreyevich Radul. 2015. Automatic differentiation in machine learning: a survey. *CoRR* abs/1502.05767 (2015). http://arxiv.org/abs/1502.05767

[7] Alvin Cheung, Shoaib Kamil, and Armando Lezama. 2015. Bridging the Gap Between General-Purpose and Domain-Specific Compilers with Synthesis. (01 2015).

[8] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[9] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets using Input-Output Examples. In *PoPL'11, January 26-28, 2011, Austin, Texas, USA* (popl'11, january 26-28, 2011, austin, texas, usa ed.). https://www.microsoft.com/en-us/research/publication/automating-string-processing-spreadsheets-using-input-output-exa

[10] Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *PLDI '14, June 09 - 11 2014, Edinburgh, United Kingdom* (pldi 2014, june 09 - 11 2014, edinburgh, united kingdom ed.). https://www.microsoft.com/en-us/research/publication/flashextract-framework-data-extraction-examples/

[11] Stratos Idreos, Konstantinos Zoumpatianos, Brian Hentschel, Michael S. Kester, and Demi Guo. 2018. The Data Calculator: Data Structure Design and Cost Synthesis From First Principles, and Learned Cost Models. In *ACM SIGMOD International Conference on Management of Data*.

[12] Ewa Luger and Abigail Sellen. 2016. "Like Having a Really Bad PA": The Gulf Between User Expectation and Experience of Conversational Agents. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 5286–5297. DOI: http://dx.doi.org/10.1145/2858036.2858288

[13] Brad A. Myers, Andrew J. Ko, and Margaret M. Burnett. 2006. Invited Research Overview: End-user Programming. In *CHI '06 Extended Abstracts on Human Factors in Computing Systems (CHI EA '06)*. ACM, New York, NY, USA, 75–80. DOI: http://dx.doi.org/10.1145/1125451.1125472

[14] Donald A. Norman. 2013. *The design of everyday things* (revised and expanded edition ed.). Basic Books.

[15] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 619–630. DOI: http://dx.doi.org/10.1145/2737924.2738007

[16] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[17] Alex Polozov and Sumit Gulwani. 2015a. FlashMeta: A Framework for Inductive Program Synthesis. In *ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH) 2015* (acm sigplan conference on systems, programming, languages and applications: software for humanity (splash) 2015 ed.). https://www.microsoft.com/en-us/research/publication/flashmeta-a-framework-for-inductive-program-synthesis/

[18] Oleksandr Polozov and Sumit Gulwani. 2015b. Microsoft PROSE SDK: A Framework for Inductive Program Synthesis. (2015).

[19] Emina Torlak and Rastislav Bodik. 2013. Growing Solver-aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. ACM, New York, NY, USA, 135–152. DOI: http://dx.doi.org/10.1145/2509578.2509586

[20] Ronald J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning* 8, 3 (01 May 1992), 229–256. DOI: http://dx.doi.org/10.1007/BF00992696