Programming by example: efficient, but not “helpful”

Mark Santolucito
Yale University
New Haven, CT, USA
mark.santolucito@yale.edu
https://orcid.org/0000-0002-1825-0097

Drew Goldman
Roslyn High School
Roslyn, NY, USA
dgoldman19@roslynschools.org

Allyson Weseley
Roslyn High School
Roslyn, NY, USA
aweseley@roslynschools.org

Ruzica Piskac
Yale University
New Haven, CT, USA
ruzica.pikac@yale.edu

Abstract

Programming by example (PBE) is a powerful programming paradigm based on example driven synthesis. Users can provide examples, and a tool automatically constructs a program that satisfies the examples. To investigate the impact of PBE on real-world users, we built a study around StriSynth, a tool for shell scripting by example, and recruited 27 working IT professionals to participate. In our study we asked the users to complete three tasks with StriSynth, and the same three tasks with PowerShell, a traditional scripting language. We found that, although our participants completed the tasks more quickly with StriSynth, they reported that they believed PowerShell to be a more helpful tool.

2012 ACM Subject Classification Human-centered computing → Empirical studies in interaction design

Keywords and phrases User Study, Scripting, Programming by Example

Digital Object Identifier 10.4230/OASIcs.PLATEAU.2018.3

Funding This research sponsored by NSF grants CCF-1302327 and CCF-1715387.

1 Introduction

Scripting languages, such as PowerShell and bash, help IT professionals to more efficiently complete tedious and repetitive tasks. Those tasks can include file manipulations and organizing data, where a simple error can destroy users' data. As an example, consider the disastrous attempt to remove all backup emacs files with the command `rm * ~`. Additionally, small errors in scripts can lead to malicious behavior, such as data loss [25]. Scripts can be difficult for users to write by hand, requiring users to have extensive experience with regular expressions, programming, and domain expertise in the scripting language of their

9th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2018).
Editors: Titus Barik, Joshua Sunshine, and Sarah Chasins; Article No. 3; pp. 3:1–3:11
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

choice. Depending on the application, a user may need to be able to write a very complicated regular expression for a relatively simple task. Furthermore, users may not have access to their scripting language of choice, depending on the operating system and software policies used by their employer.

For these reasons, many end-users search for help on online forums when they need to write a script [3, 2, 4]. When users seek help in writing a script on forums, they will often provide a few illustrative examples that convey the goal of the script. This observation was the basis of StriSynth [12], a research tool that was proposed to make scripting easier and more efficient by allowing users to program scripts by example. While scripting is a challenging task, especially for novice programmers, providing examples of the intended behavior is a more natural interface for scripting. StriSynth supports various types of functions, such as transformations, filters, partitions, and merging strings.

In this work, we explore how scripting by example, specifically with StriSynth, is received by the real-world target end-users. We designed a user study around StriSynth and recruited 27 IT professionals to participate in the study. In our study we asked users to complete three tasks with StriSynth, and the same three tasks with PowerShell, a traditional scripting language. When using StriSynth, users were statistically significantly faster at completing tasks as compared with PowerShell. However, in a post-study survey when users were asked which tool they perceived to be more "helpful", users statistically significantly reported that PowerShell, with the traditional scripting paradigm, was more helpful. This was counter-intuitive result, as we expected that a faster tool should be considered to be more helpful by users. While the formal methods community has largely taken efficiency of task completion to be an indicator of a good language design, we explore our results here that show this is in fact a more complex issue.

2 Background

Programming by example (PBE) [6, 23, 27, 7] is a form of program synthesis. It works by automatically generating programs that coincide with the given examples. In this way, the examples can be seen as an incomplete, but easily readable and understandable specification. However, even if the synthesized program satisfies all the provided examples, it might not correspond to user's intentions due to this incompleteness in the specification. In this case, a user must provide further examples to the synthesis tool.

To address this issue, StriSynth was implemented as a live programming environment [5] for PBE. In this way, a synthesized script can be refined with every new provided example, and thus yields a more interactive experience for the user. Interactive PBE allows end-users to provide a single example at a time, rather than guessing at the full example set that is necessary for synthesis.

In order to compare the PBE paradigm to more traditional scripting languages, we have chosen to use the tool StriSynth [12]. StriSynth is an existing tool for automating file manipulation tasks, in a similar style to Flash Fill's [1] synthesis of spreadsheet manipulations. While the use of scripting language such as sed, awk, Bash or PowerShell requires a certain level of expertise, many tasks can be easily described using natural language or through examples.

2.1 StriSynth example

To give some context for how StriSynth compares to traditional scripting language paradigms, we give an example task that can be easily completed with StriSynth. This task comes from a StackOverflow post, where the users discuss challenging regular expressions [3]. The user asked for a script that will create a link from every item in a directory. To better illustrate the goal of the script, the user provided two examples transformations:

```
Document1.docx          <A HREF='Document1.docx'>Document1</A>
Document2.docx    ⇨     <A HREF='Document2.docx'>Document2</A>
```

To accomplish this transformation, other users on the forum suggested a solution based on regular expressions in `sed`:

```
sed/\(^[a-zA-Z0-9]+\)\.\([a-z]+\)/\<a href\=\'\1\.\2\' \>\1\<\/a\>/g
```

While it was very easy for the user to express the goal of the script by providing examples, the resulting script is arguably less readable, even for such a simple problem. In contrast, to solve this problem in StriSynth, a user provides an example showing what a script should do:

```
> NEW
> "f.docx" ==> "<a href='f.docx'>a</a>"
> val F = TRANSFORM
```

The keyword `NEW` denotes the start for learning of a new script, after which the user provides an example of the scripts desired behavior. Based on the provided example, StriSynth learns a string transformer, and the user saves it with the next command. Every learned function can be saved using the command `val name = ...` which creates a reference, `name`, to the learned script. The user may then check how F works on different examples to confirm the learned function is correct.

```
> F("Document1.docx")
<A HREF='Document1.docx'>Document1</A>
> F("Document2.docx")
<A HREF='Document2.docx'>Document2</A>
```

We observe that the learned transformer F is a function that exactly does what the user asked initially. However, it only takes a single string as input, while the user wanted a script that operates on a list of strings. To extend the learned transformer to work over a list, the user can use the `as map` function.

```
> val finalScript = F as map
```

If a function G has a type signature $G : T_1 \rightarrow T_2$, then applying the postfix operator `as map` will result in G `as map` $: \text{List}(T_1) \rightarrow \text{List}(T_2)$. With `as map`, the user creates the final script which takes as input a list of file names and creates a list of HTML links.

Beyond the string transformation used above, StriSynth can also learn other types of functions from examples. StriSynth supports a *filter* function that takes a list of strings as input and removes some elements based on the filtering criterion. Similarly, StriSynth also supports learning a *partition* function takes as input a list of strings, and divides them into groups based on the partitioning criterion. Those groups are then returned as a list of lists of strings. This functions can be used in any way by the user, but are particularly useful for scripting tasks that require operations on certain types of files, or files matching some naming pattern.

In addition, StriSynth can learn a *reduce* function that merges the elements in a list into a single string. StriSynth's *split* function does the opposite: it returns a list of strings from the input string. These types of functions are especially useful for scripting tasks that apply operations to collections of files.

3 Methodology

A recent survey of the key challenges facing formal methods cites the need for more user studies, especially on real-world users [15]. To test the impact PBE on real users, we recruited 27 IT professionals, all of whom were 18 years of age or older. All materials for the study, as well as the raw data results from the study are available open source at `https://github.com/santolucito/StriSynthStudy`.

Our study design consisted of four stages:

1. A tutorial on both PowerShell and StriSynth that introduced the paradigm and syntax
2. Complete three scripting tasks (*Extract filenames* from a directory listing, *Move files* with `*.png` to `imgs/`, *Printing pdfs* from a list of various file types) in PowerShell
3. Complete the same three scripting tasks in StriSynth
4. A post-study survey

In the study, participants were told that they would be using the tools StriSynthA and StriSynthB instead of StriSynth and PowerShell to avoid bias from participants' prior experience. The participants were randomly split into two groups, group A and group B, where the two groups switched the order of steps 2 and 3 of the study to account for any potential bias in earlier exposure to the tasks. Group A completed the tasks with PowerShell first (N=12) and group B completed the tasks with StriSynth first (N=15). The entire study generally took each participant 50 minutes, and the study was conducted in-person with a researcher present. The scripting tasks were completed on the researcher's laptop, which was preloaded before each study with directories and files needed for the scripting tasks.

While each user was participating in the study, the researcher present recorded the overall time that was used to complete each task. Following the completion of the six tasks, each user was given a questionnaire. The questionnaire measured various responses: prior coding experience, perceived helpfulness of each program as a whole, and perceived helpfulness of each program for each specific task they completed.

4 Results

In this section we present the results of the user study described in Sec. 3. Overall, users completed the tasks more quickly when using StriSynth as opposed to PowerShell. This is good evidence that StriSynth is an efficient tool, especially as none of our users had used StriSynth before this study, while some already had experience with PowerShell. However, despite this concrete measure of efficiency for StriSynth, users said that they believe that PowerShell is a more helpful tool.

4.1 Time to complete the user study tasks

To estimate the usefulness of the programming by example tool StriSynth, we recorded the time it took for users to complete each task with both StriSynth and PowerShell. The results are shown in Fig. 1. In addition, Fig. 1 also contains standard error, depicted with line bars.

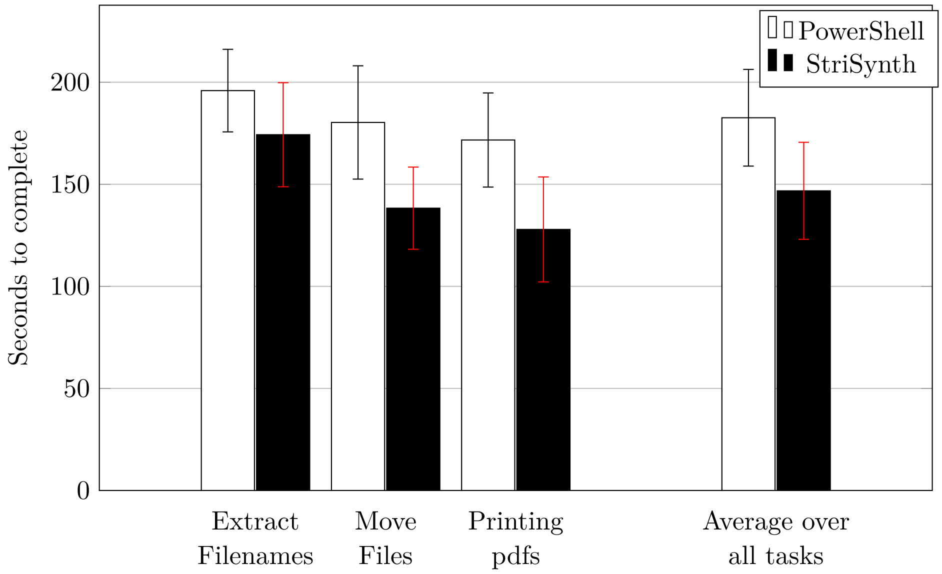

Figure 1 The amount of the time each task took, as well as the average time over all tasks for all users (N=27). The smaller bars indicate standard error.

In the case of the first task (extracting filenames), from in Fig. 1 the standard error bars give us the intuition that true mean of the time it takes for overall population to complete this task using PowerShell is between 170 and 210 seconds. The smaller the standard error, the more likely is that we have achieved the exact, true value of the mean time, which it takes for the entire population of IT professionals to complete the tasks.

We can see in Fig. 1 that overall the users took less time to complete the tasks with StriSynth. However, our sample size was relatively small (N=27). Therefore, we wanted to measure the confidence that our observations are reflective of the larger IT population beyond our small sample size. To do this we ran a paired sample t-test [32].

When running the paired sample t-test, we are checking the null hypothesis that the difference between the paired observations in the two samples is zero. Without going into the details of statistical methods, we need to compute the p-value. Any p-value of less than .05 is called *statistically significant*, indicating we have met a generally accepted threshold of confidence in our results [32].

By running these tests on our samples, we learn that a statistically significant difference was found in the *Move Files* ($p = .03$) and *Printing pdfs* ($p = .02$) tasks. The p-value of .03 means that, assuming StriSynth does *not* actually have any impact on time to complete the *Move Files* task, there is only a 3% chance that we could have observed the timing difference (or even some larger difference) between StriSynth and PowerShell presented Fig. 1. In other words, given these low p-values, we can be confident that using StriSynth does in fact have an impact on time to complete the task.

All together, our results support the claim that, for small scripting tasks of the type we presented to our users, PBE can be a more efficient programming paradigm. This is the expected result that is in line with the literature [16].

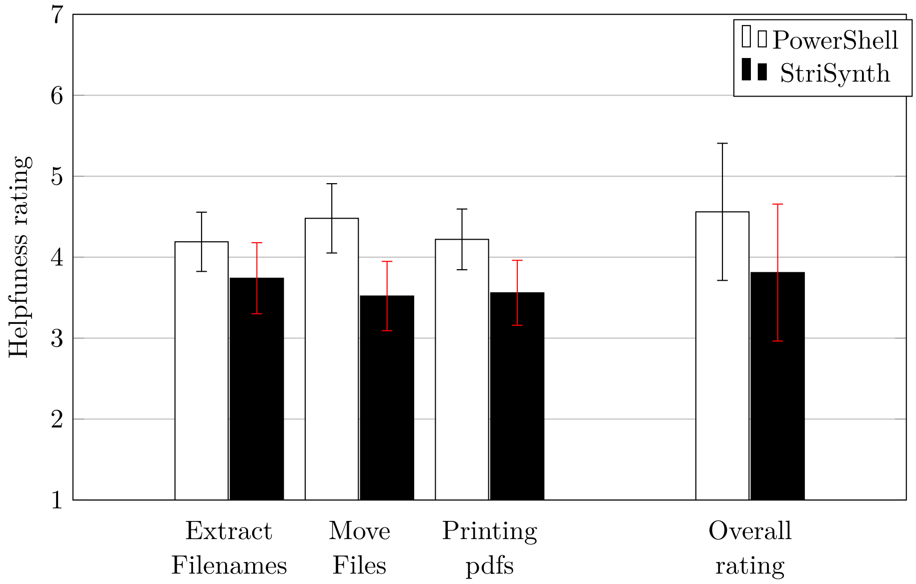

Figure 2 Users' (N=27) self reported measure of the helpfulness of each tool with standard error bars.

4.2 Reported helpfulness

Reaching beyond traditional measures for PBE, at the end of the study we also asked users to report how "helpful" they found both StriSynth and PowerShell. At this point, users did not know how long they took to complete the tasks with each of the tools. Users were asked the rate the helpfulness only based on their experience of using the tools during the study. The exact questions asked were "The following program was helpful for scripting/completing *Extract Filenames*/etc...", and users were asked to respond on a scale from 1 (strongly disagree) - 7 (strongly agree). We show the results from this survey question in Fig. 2, again with standard error bars. Users rated PowerShell as more helpful in all three tasks, with the *Move Files* task showing the most significant difference ($p < .01$).

The results in Fig. 2 show the surprising insight that, despite the efficiency of StriSynth as demonstrated in Fig. 1, users perceived PowerShell to be the more helpful tool. Unfortunately, as we did not anticipate such unexpected results, our study design did not include a more detailed definition of helpfulness, or ask users to give a more detailed description of their interpretation of what it means for a tool to be helpful. However, we can at least surmise from the results presented here, that efficiency is not a complete proxy measure for helpfulness of a tool.

4.3 Impact of prior user experience

Prior work observed that familiarity can be a stronger indicator of user preference than efficiency in the development of programming languages [31]. Our study asked users to self-report their prior experience with scripting languages in a post-study survey to understand the impact of user familiarity. The survey used a seven-point Likert scale for users assess the users' prior experience. Fig. 3 shows the distribution of experience in three categories for all users.

To understand the impact of prior experience on how the users interacted with StriSynth,

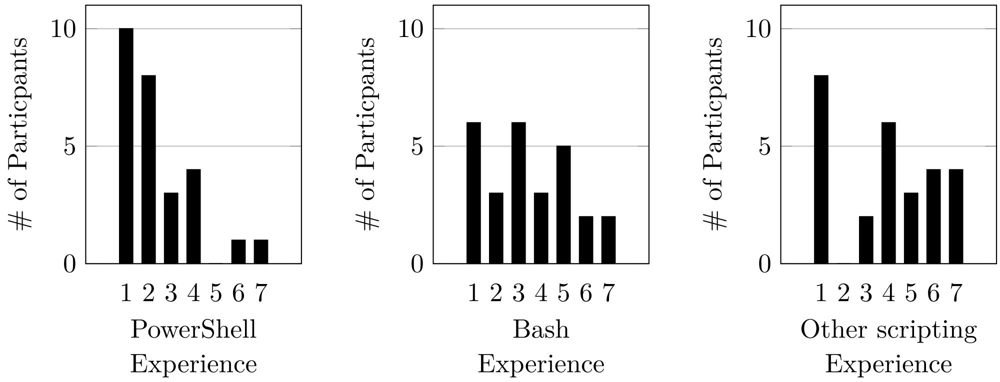

Figure 3 Users' (N=27) self reported prior experience with various scripting languages from 1 (Unfamiliar) to 7 (Expert User).

we split our user population into two categories. We have the inexperienced user group, which is the users who rated their prior experience with PowerShell as a 1 (unfamiliar), and the complement set of users as the experienced user group, who rated their prior experience with PowerShell as (≥ 2). In Fig. 4, we show how these two groups performed in the study.

Fig. 4 shows that both groups of users completed the tasks faster with StriSynth. A more subtle and interesting insight is that inexperienced users had a greater relative speedup in task completion when using StriSynth. That is, inexperienced users benefited more from using StriSynth as compared to the benefit to experienced users. This provides evidence for the widely stated perception that programming by example is a domain well-suited for novice programmers.

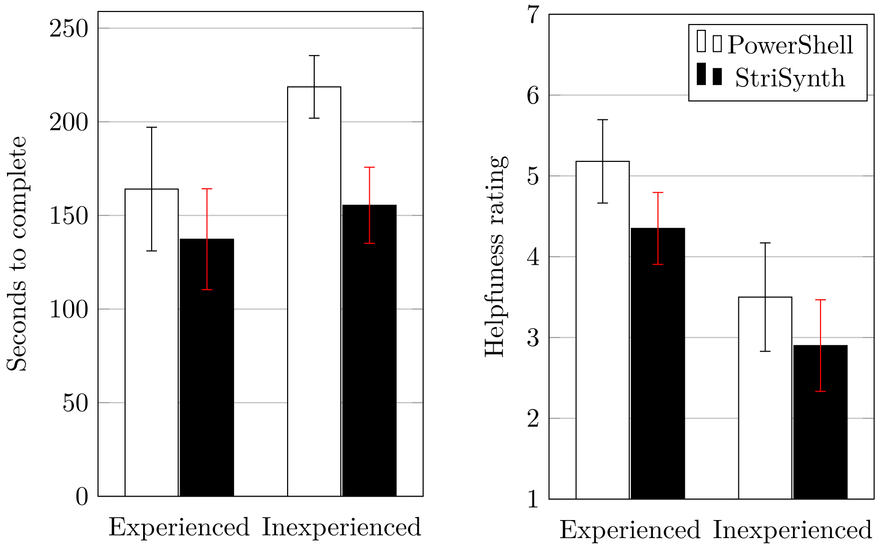

Figure 4 We grouped users as Experienced (PowerShell experience≥2, N=17) and Inexperienced (PowerShell experience=1, N=10). We report average time to complete the tasks, and self reported helpfulness of the tools, as separated by these two groups.

4.4 Threats to Validity

In a usability study, it is important to avoid any possible selection bias in the call for participants. Selection bias can be an issue if the set of users selected systematically differs from the target population. The results we have presented are from a set of users that work as professional IT support specialists. We do not believe that we have any selection bias here because in this work, we specifically wanted to explore the impact synthesis can have in the real-world on such professionals.

A further potential threat to the validity of our results is in the social desirability bias, or need-to-please phenomena, whereby users will subconsciously try to produce the results they expect the researcher would like to see. This potential bias can occur when users are asked to compare a tool that is a known standard with an alternative that the user knows to be developed by the researcher. To combat this issue, we presented StriSynth and PowerShell as tools named StriSynthA and StriSynthB respectively. In this way, we framed the study as a comparison between two different tools that we had developed, eliminating the potential need-to-please bias. This was a critical component to our study design that allowed us to observe the disconnect between efficiency and users' perceived helpfulness of each tool.

5 Discussion

A key question from these results remains - how are efficiency and helpfulness different in the eyes of the users, and why is this difference manifest in our study? One tempting explanation is that this is the result of two vastly different interfaces. PowerShell is an industrially developed tool, while StriSynth is a research prototype. However, both are command line utilities with a qualitatively similar user experience. Another possible point of departure is in the ability of the user to understand the function of a synthesized script from StriSynth. Trust in the result of program synthesis is a direction that needs further exploration, but StriSynth is unique in this respect in that it provides an English text explanation of the synthesis result. Another possible interpretation may be tied to the expressivity of the paradigm - StriSynth and other PBE tools are generally limited in their ability to directly work with a traditional programming language and use familiar concepts such as variables and loops. This may make a language seem less helpful for new users.

Finally, the results from our user study are specifically targeted at the impact of programming by example systems for scripting in IT professional populations. We must also consider how our results can be interpreted and extended to other PBE domains and program synthesis more generally.

5.1 Application to Related Work

Gulwani et al. [11] show that PBE is an effective paradigm for industrial application in spreadsheet manipulation, such as string transformations [1, 9], table transformations [10] and database look-ups [29]. Another approach is based on the abstraction of 'topes' [28], which lets users create abstractions for different data present in a spreadsheet. With topes, a programmer uses a GUI to define constraints on the data, and to generate a context-free grammar that is used to validate and reformat the data. These application domains of PBE are focused on a similar population of non-expert programmers, and so it may be possible to observe a similar efficiency vs helpfulness phenomena.

Unlike programming by example, in which the user provides input-output examples, programming by demonstration is characterized by the user providing a complete trace demonstration leading from the initial to the final state. There are several programming by demonstration systems [6], such as Simultaneous Editing [26] for string manipulation, SMARTedit [22] for text manipulation and Wrangler [17] for table transformations. As programming by demonstration requires intermediate configurations instead of just input and output examples, this paradigm is usually less flexible [21] than programming by example, but the synthesis problem is easier. Based on our results here, it is possible that this reduced flexibility may indicate users would rate programming by demonstration even less helpful (but possible more efficient) than PBE in certain domains. There has been work to overcome the limited expressivity of programming by demonstration by combining program synthesis with direct manipulation of output [24]. This approach may present a way to resolve the disconnect between efficiency and helpfulness as it allows users to interact in both a traditional programming style as well as with synthesis.

The Myth [27] and Λ^2 [7] systems support PBE for inductively defined data-types in functional languages. In contrast to StriSynth which focuses on scripting tasks, these tools are focused on synthesis for more general purpose programming languages. The results from our study may be cautiously extrapolated other domains - while the theme of PBE is the same, interaction preference for users may differ when looking at general purpose languages.

Instead of providing specification in terms of examples or demonstrations, specification can also be given in more formal and complete ways. InSynth [14, 13], CodeHint [8] and the C# code snippets on demand [33] are systems that aim to provide code snippets based on context such as the inferred type or the surrounding comments. Leon [20] and Comfusy [19, 18] synthesize code snippets based on complete specifications, which are written in the same language that is used for programming. Sketch [30] takes as input an incomplete program with holes, and synthesizes code to complete the so that it meets the specification. These techniques provide a more nuanced interface that may seem, from a perspective of helpfulness, to be more similar to a traditional language paradigm.

6 Conclusions

Our study shows that users do not always correlate an efficient programming paradigm with a helpful paradigm. A more thorough exploration of this finding requires a follow up study, in particular to discover the definition of helpfulness that participants are using. A key question to answer would be whether users had erroneously perceived PowerShell to be more efficient and therefore helpful, or if users consciously have other metrics in mind that constitute the helpfulness of programming paradigm.

References

1 Flash Fill (Microsoft Excel 2013 feature). `http://research.microsoft.com/users/sumitg/flashfill.html`.

2 Stack Overflow: Auto increment a variable in regex. `http://goo.gl/GPuZP3`. Accessed: 2015-03-25.

3 Stack Overflow: What is the most difficult/challenging regular expression you have ever written? `http://goo.gl/LLJe0r`. Accessed: 2015-03-24.

4 Super User: How to batch combine jpeg's from folders into pdf's? `http://goo.gl/LnGYH7`. Accessed: 2015-05-13.

5 Sebastian Burckhardt, Manuel Fähndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. It's alive! Continuous feedback in UI programming. In *PLDI*, 2013.

6 A. Cypher and D.C. Halbert. *Watch what I Do: Programming by Demonstration.* MIT Press, 1993.

7 John Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, 2015.

8 Joel Galenson, Philip Reames, Rastislav Bodík, Björn Hartmann, and Koushik Sen. Code-Hint: dynamic and interactive synthesis of code snippets. In *ICSE*, 2014.

9 Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.

10 Sumit Gulwani. Synthesis from examples: Interaction models and algorithms. In *SYNASC*, 2012.

11 Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8), 2012.

12 Sumit Gulwani, Mikael Mayer, Filip Niksic, and Ruzica Piskac. Strisynth: Synthesis for live programming. In *ICSE*, 2015.

13 Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *PLDI*, 2013.

14 Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. Interactive synthesis of code snippets. In *CAV*, 2011.

15 Reiner Hähnle and Marieke Huisman. 24 challenges in deductive software verification. In Giles Reger and Dmitriy Traytel, editors, *ARCADE 2017. 1st International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements*, volume 51 of *EPiC Series in Computing*. EasyChair, 2017.

16 Zhongjun Jin, Michael R. Anderson, Michael Cafarella, and H. V. Jagadish. Foofah: Transforming data by example. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD ’17. ACM, 2017.

17 Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. Wrangler: interactive visual specification of data transformation scripts. In *CHI*, 2011.

18 Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Comfusy: A tool for complete functional synthesis. In *CAV*, 2010.

19 Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *PLDI*, 2010.

20 Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Software synthesis procedures. *Commun. ACM*, 55(2), 2012.

21 Tessa Lau. Why programming-by-demonstration systems fail: Lessons learned for usable AI. *AI Magazine*, 30(4), 2009.

22 Tessa A. Lau, Pedro Domingos, and Daniel S. Weld. Learning programs from traces using version space algebra. In *K-CAP*, 2003.

23 H. Lieberman. *Your Wish Is My Command: Programming by Example.* Morgan Kaufmann, 2001.

24 Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. Bidirectional evaluation with direct manipulation. *PACMPL*, 2(OOPSLA), 2018.

25 Karl Mazurak and Steve Zdancewic. Abash: Finding bugs in bash scripts. In *In ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 2007.

26 Robert C. Miller and Brad A. Myers. Interactive simultaneous editing of multiple text regions. In *USENIX*, 2001.

27 Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, 2015.

28 Christopher Scaffidi, Brad A. Myers, and Mary Shaw. Topes: reusable abstractions for validating data. In *ICSE*, 2008.

29 Rishabh Singh and Sumit Gulwani. Learning semantic string transformations from examples. *PVLDB*, 5(8), 2012.

30 Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, UC Berkeley, 2008.

31 Joshua Sunshine. *Protocol Programmability*. PhD thesis, Pittsburgh, PA, USA, 2013.

32 Ronald E Walpole, Raymond H Myers, Sharon L Myers, and Keying Ye. *Probability and statistics for engineers and scientists*, volume 5. Macmillan New York, 1993.

33 Yi Wei, Youssef Hamadi, Sumit Gulwani, and Mukund Raghothaman. C# code snippets on-demand, 2014. `http://codesnippet.research.microsoft.com`.