User Interaction Models for Disambiguation in Programming by Example

Mikaël Mayer[*] mikael.mayer@epfl.ch
Gustavo Soares[†] gsoares@dsc.ufcg.edu.br
Maxim Grechkin[‡] grechkin@cs.washington.edu
Vu Le[§] vmle@ucdavis.edu
Mark Marron[¶] marron
Oleksandr Polozov[‡] polozov@cs.washington.edu
Rishabh Singh[¶] risin
Benjamin Zorn[¶] zorn
Sumit Gulwani[¶] sumitg

ABSTRACT

Programming by Examples (PBE) has the potential to revolutionize end-user programming by enabling end users, most of whom are non-programmers, to create small scripts for automating repetitive tasks. However, examples, though often easy to provide, are an ambiguous specification of the user's intent. Because of that, a key impedance in adoption of PBE systems is the lack of user confidence in the correctness of the program that was synthesized by the system. We present two novel user interaction models that communicate actionable information to the user to help resolve ambiguity in the examples. One of these models allows the user to effectively navigate between the huge set of programs that are consistent with the examples provided by the user. The other model uses active learning to ask directed example-based questions to the user on the test input data over which the user intends to run the synthesized program. Our user studies show that each of these models significantly reduces the number of errors in the performed task without any difference in completion time. Moreover, both models are perceived as useful, and the proactive active-learning based model has a slightly higher preference regarding the users' confidence in the result.

INTRODUCTION

Today, billions of users have access to computational devices. However, 99% of these end users do not have programming expertise and they often struggle with repetitive tasks in various domains that could otherwise be automated using small scripts. Programming-by-examples (PBE) [19, 5] has the potential to revolutionize this landscape since users can often specify their intent using examples as has been observed on various help forums [8]. PBE involves techniques that generalize example behaviors on concrete inputs provided by the user into programs that can operate on new unseen inputs. PBE has traditionally been applied to synthesizing small programs in various domain-specific languages (DSLs) such as string and table transformations [8] and data extraction [17]. PBE has been pursued in various communities including programming languages [18, 6, 4], inductive programming [9], machine learning [21], artificial intelligence [28], and databases [30]. Work in these communities has focused on addressing one of the key challenges in PBE, that of efficiently searching the huge state space (potentially infinite) of programs defined by the underlying DSL for a program that is consistent with the user-provided examples.

However, not much attention has been given to dealing with another key technical challenge in PBE, that of dealing with ambiguities. Examples are an ambiguous form of specification in the sense that there can be different programs that are consistent with the provided examples, but these programs differ in their behavior on some other inputs. The underlying PBE system might end up synthesizing an unintended program that is consistent with the examples provided by the user but does not generate the intended results on some other inputs that the user cares about. In 2009 Tessa Lau presented a critical discussion of PBE systems noting that adoption of PBE systems is not yet widespread, and proposing that this is mainly due to lack of usability and confidence in such systems [14]. complementary user interaction models for PBE that help increase user confidence in the underlying system.

Motivational Real-world PBE Case Studies

Recently, a first mass-market PBE product was released in the form of the *FlashFill* feature in Microsoft Excel 2013. It allows end users to automate sophisticated string transformations in real time from one or more user-provided examples [7]. While the PBE engine behind FlashFill received many positive reviews from popular media (`bit.ly/flashfill`) the user interface for FlashFill leaves a lot to be desired. John Walkenbach, an author renowned for his Excel textbooks, labeled FlashFill as a "controversial" feature. He wrote "It's a great concept, but it can also lead to lots of bad data. (...) Be very careful. (...) [M]ost of the extracted data will be fine. But there might be exceptions that you don't notice unless you examine the results very carefully." (`spreadsheetpage.com/index.php/blog/C10`)

Another mass-market PBE product, recently released as part of the Windows 10 preview, is the *ConvertFrom-String* feature in PowerShell (`bit.ly/convertfrom-string`). It allows end users to extract structured data out of semi-structured text/log files from one or more user-provided examples. It is based on the FlashExtract PBE engine that can synthesize

[*] EPFL, Switzerland
[†] UFCG, Brazil
[‡] UW, Seattle, WA, US
[§] UC Davis, CA, US
[¶] Microsoft Research, Redmond, WA. Email: ...@microsoft.com

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
UIST '15, November 08-11, 2015, Charlotte, NC, USA
© 2015 ACM. ISBN 978-1-4503-3779-3/15/11$15.00
DOI: http://dx.doi.org/10.1145/2807442.2807459

sophisticated data extraction scripts in real time [17]. It was well-received by various Microsoft MVPs (Most Valued Professionals), who described it as "New kid on the block", "This is super cool !!", "must admit that this cmdlet is to me one of the best improvement that came with WMF5.0 and PowerShell v5". (`bit.ly/flashextract`) However, the MVPs also complained that they had no visibility into the process for debugging purposes. This prompted Microsoft to release an improved version of FlashExtract that provided a flag to display the top-ranked program synthesized by FlashExtract. An MVP still complained: "If you can understand this, you're a better person than I am."

User Interaction Models

We propose two novel user interaction models that aim to alleviate above-mentioned transparency concerns by exposing more information to the user in a form that can be easily understood and acted upon. These models help resolve ambiguity in the example-based specification, thereby increasing user's trust in the results produced by the PBE engine.

Program Navigation: A typical PBE engine operates by synthesizing multiple programs that are consistent with the examples provided by the user, and then ranking the programs in order of their likelihood of being the intended program [8]. A typical PBE interface would pick the top-ranked program and use it to automate the user's task; possibly this top-ranked program can even be shown to the user. We propose a novel user interaction model, called Program Navigation, that allows the user to navigate between all programs synthesized by the underlying PBE engine (as opposed to displaying only the top-ranked program) and to pick one that is intended. The number of such programs can usually be huge (several powers of 10 such as 10^{30} [8]). However, these programs usually share common sub-expressions and are described succinctly using version space algebra based data structures [27]. We leverage this sharing to create a navigational interface that allows the user to select from different ranked choices for various parts of the top-ranked program. Furthermore, these programs are paraphrased in English for easy readability.

Conversational Clarification: We propose a complementary novel user interaction model based on active learning, called Conversational Clarification, wherein the system asks questions to the user to resolve ambiguities in the user's specification with respect to the available test data. These questions are generated after the PBE engine has synthesized multiple programs that are consistent with the user-provided examples. The system executes these multiple programs on the test data to identify any discrepancies in the execution and uses that as the basis for asking questions to the user. The user responses are used to refine the initial example-based specification and the process of program synthesis is repeated.

FlashProg *Framework for Data Manipulation*

We have implemented the above two user interaction models in a generic manner in a UI framework called FlashProg. The FlashProg framework provides UI support for several PBE engines related to data manipulation, namely FlashFill [7], FlashRelate [4], FlashExtract [17], and FlashWeb. Even though PBE has been applied to various application domains, we focus our attention in this paper on data manipulation, which we believe is one of the most impactful applications for PBE. Data is locked up in semi-structured formats (such as spreadsheets, text/log files, webpages, and PDF documents), which offer great flexibility in storing hierarchical data by combining presentation/formatting with the underlying data model, but make it extremely hard to manipulate that data. PBE holds the promise of enabling a delightful data wrangling experience because many tedious data manipulation tasks such as extraction, transformation, and formatting can be easily described using examples.

The FlashProg UI builds over the STEPS approach [32] to PBE, wherein the user breaks down a sophisticated task into a sequence of simpler steps, and each step is automated using PBE. We conducted a user study, where we asked participants to extract structured data from semi-structured text files using FlashProg. We observe that participants perform more correct extraction when they make use of the new interaction models. To our surprise, participants preferred Conversational Clarification over Program Navigation slightly more even though past case studies suggested that users wanted to look at the synthesized programs. We believe this is because Conversational Clarification is a *proactive* interface that asks clarifying questions, whereas Program Navigation is a *reactive* interface that expects an explicit correction of a mistake. This paper makes the following contributions:

- We propose a user interaction model for PBE called Program Navigation. It lets the users browse the large space of programs that satisfy the user specification by selecting ranked alternatives for different program subexpressions.
- We propose another complementary user interaction model for PBE called Conversational Clarification. It involves asking directed example-based questions to the user, whose responses are then automatically fed back into the example-based specification model.
- We present a generic framework called FlashProg that implements Program Navigation and Conversational Clarification on top of any PBE engine. We have used FlashProg to develop user interfaces for four different PBE engines.
- We present results of a user study that evaluated our two user interaction models. We discover that both models significantly reduce the number of errors without any difference in completion time. Both models are perceived as useful, but Conversational Clarification has a slightly higher preference w.r.t. the users' confidence in the result.

RELATED WORK

FlashProg user interface is inspired by that of the STEPS system [32] that uses hierarchical structure coloring for text extraction and manipulation. STEPS showed the usefulness of PBE systems for text processing: STEPS users completed more tasks and were faster than conventional programmers. For disambiguation and converging to the desired task, STEPS supports two interaction mechanisms: (i) provide additional mock input-output examples that capture specific intents and corner cases, and (ii) navigate through a flattened list of a small set of programs (paraphrased in English). Since the DSLs supported by FlashProg are more expressive, there is often a huge number of programs that are consistent with few examples, which makes the interaction model of navigating the flattened list of programs unusable. Providing mock input-output examples puts additional burden on users to first identify why the system is learning an incorrect program and then construct specific examples to avoid learning them. FlashProg provides two new interaction models to alleviate this problem: 1) Program Navigation to browse the set of learned programs (paraphrased in English) in a hierarchical manner, and 2) Conversational Clarification to ask users to select the desired output on inputs for which the system has learned multiple interpretations.

Wrangler [12] is an interactive system for data transformations on tabular data. It automatically suggests a ranked list of paraphrased transformations based on the context of user interactions. A user can then navigate the space of suggested transformations in three ways: (i) by providing additional examples, (ii) by selecting an operator from the transform menu, and (iii) by editing the parameters of the suggested transforms. Wrangler's language is aimed at data cleaning and transformation, but not for extracting data from semi-structured sources. Moreover, the new interaction models of Program Navigation and Conversational Clarification can augment and complement Wrangler's interaction model.

LAPIS [23] is a text-editor that incorporates the concept of *lightweight structure* to recognize the text structure using an extensible library of patterns and parsers. Given positive and negative examples, LAPIS learns a pattern in a language called *text constraints* (TC), and highlights other matches in the file. This enables users to perform multiple selections and simultaneous editing to apply the same set of edits to a group of elements. LAPIS does not has good support for nested and overlapping regions, which are essential for data extraction tasks. LAPIS also introduced the idea of outlier detection for finding atypical pattern matches to focus user's attention for potential incorrect generalizations [22], which is related to the Conversational Clarification interaction model. The main difference between the two is the way in which the match discrepancies are computed. LAPIS models pattern matches as a list of binary-valued features and computes outlier matches based on their weighted Euclidean distance from the feature vector of the median match. FlashProg uses program semantics to identify ambiguous examples, where the highly ranked learnt programs generate different outputs on the examples.

Amershi et al. [2, 3] have explored two strategies for soliciting effective training examples in interactive ML systems. The first strategy of *global overview* selects a subset of training examples that maximizes the mutual information with the high-dimensional vector space of the examples, and is most representative of the training set. The second strategy of *projected overview* projects examples onto a set of principal dimensions and then selects examples that illustrate variation amongst those dimensions. Our Conversational Clarification model presents a complimentary technique for selecting training examples to learn a richer class of programs (as opposed to classifiers) based on the semantics of the learnt programs.

Several PBE-based text manipulation systems exist. FlashFill [7] learns syntactic string transformations (involving concatenation of regex-based substrings) from few examples. SmartEdit [16] automates text processing tasks from demonstrations by interactively navigating the space of learned programs (represented using a version-space algebra) using a mixed-initiative interface. Visual AWK [13] provides a graphical environment to drag and drop relevant text selections to learn patterns based on trial and error demonstrations. It allows users to separately learn conditionals and edit the learned programs graphically. Peridot [25] allows users to interactively create graphical user interfaces by demonstrations. The TELS [31] system records a trace, generalizes it, and then executes and extends the generated program based on user feedback. Marquise [26] lets users provide example actions to create user interfaces and uses a feedback window to show the inferred operation using english sentences with buttons that can be pressed to pop up the list of alternative options. Many of these systems do not expose the learned programs to the user and depend on manual inspection of generated outputs for validation. However, some systems such as SmartEdit, Peridot, Marquise, and Visual AWK do expose the learned programs, but the class of transformations supported by them are limited and are not expressive enough for learning hierarchical extraction of nested records.

FlashProg is based on automated program synthesis. Programs are synthesized in DSLs that are expressive enough to encode most common tasks, but at the same time concise enough for efficient learning. The synthesis algorithm uses a divide-and-conquer based strategy to decompose the original learning task to smaller sub-tasks [27]. This approach has been used to develop several PBE systems in the domains of syntactic string transformations [7], semantic string transformations [8], data extraction from semi-structured sources [17], and transformation of semi-structured tables [4]. The FlashProg framework provides a general user interface for all these PBE systems, where users can use Program Navigation to navigate the space of learned programs in a hierarchical manner, and use Conversational Clarification to provide additional examples.

Jha et.al. [10] proposed *distinguishing inputs* for disambiguation in program synthesis - their synthesizer generates two consistent programs P_1 and P_2, and a distinguishing input on which P_1 and P_2 yield different results. The Conversational Clarification interaction model uses a similar idea to ask questions but it differs in several ways: (i) it selects distinguishing inputs from the user data instead of generating random inputs, (ii) it converges faster since it can execute all learned programs (instead of two) to ask for *more important* clarifications, and (iii) it works in real-time and is interactive unlike the constraint-solver based technique used in [10].

Topes [29] allows developers to implement abstractions for interactively validating and transforming data in many different formats. It can recognize valid inputs in multiple different formats on a non-binary scale as opposed to binary-valued regular expressions. It provides transformation functions to convert inputs in different formats to a consistent format.

The DSLs for FlashProg build on top of regular expressions and are quite different from the validation and transformation functions supported by Topes. Conversational Clarification uses the set of learnt programs to find ambiguous inputs unlike the non-binary valued matches used by Topes for finding questionable inputs.

Gamut [20] is a PBD system that enables non-programmers to create interactive games and educational software using demonstrations. Gamut's interaction techniques allows users to specify relationships between developer-generated objects such as guide objects, cards, and decks of cards, and then use *nudges* and *hints* to modify or provide new behaviors. The ”Do Something” interaction model lets users specify new behaviors on an object, whereas the ”Stop That” interaction model lets users specify undesired behaviors. Similar to the ”Stop That” model, FlashProg also lets users specify negative examples by clicking the labelled output in the input pane or marking the entry in the output table as incorrect.

`Import.Io` and `Kimono` are recent commercial tools that aim at extracting data from semi-structured sources. `Import.Io` performs extraction automatically without any human intervention. Although this works well for some simple semi-structured sources, it fails on relatively complex data sources. Adding support for handling newer semi-structured sources would require one to add new complex rules and heuristics. Kimono, on the other hand, performs data extraction by examples similar to FlashProg and provides a similar user interface. The range of logics for extracting sub-string data from html elements supported by Kimono, however, is not as rich compared to FlashProg. The learned regular expressions exposed by Kimono are too low-level to be easily understable by programmers, whereas FlashProg paraphrases the set of learned programs in a hierarchical manner. Moreover, Kimono does not support any conversational interaction model for disambiguating ambiguous cases.

FLASHPROG USER INTERFACE

FlashProg is a web application for PBE-based data extraction from textual documents, spreadsheets, and Web pages. In this overview, we focus on the text domain, but the UI behaves similarly for all other domains as well. Figure 1 shows a FlashProg window after providing several examples (on the left), and after invoking the learning process (on the right). The FlashProg window consists of 3 sections: Top Toolbar (1), Input Text View (2), and PBE Interaction View (3).

The Top Toolbar contains: (a) an input button to open and upload files, (b) a button that resets FlashProg to an initial state, (c) undo/redo buttons as expected, and (d) a “Results” button to download the output as a CSV file for further processing.

The Input Text View is the main area. It gives users the ability to provide examples by highlighting desired sections of the document, producing a set of nested colored blocks. Additionally, users may omit the structure boundary and only provide examples for the fields as shown in Figure 1. After an automated learning phase, the output of the highest ranked program is displayed in the Output pane. Each new row in the output is also matched to the corresponding region of the original document that is highlighted with dimmer colors. The scroll bars are colored with a *bird's-eye view* of highlighting, as a minimap feature (as **SublimeText.com**). We have found this view helpful for looking for discrepancies in the produced highlighting. The user can also provide *negative examples* by clicking on previously marked regions to communicate to the PBE system that the region should not be selected as part of the output.

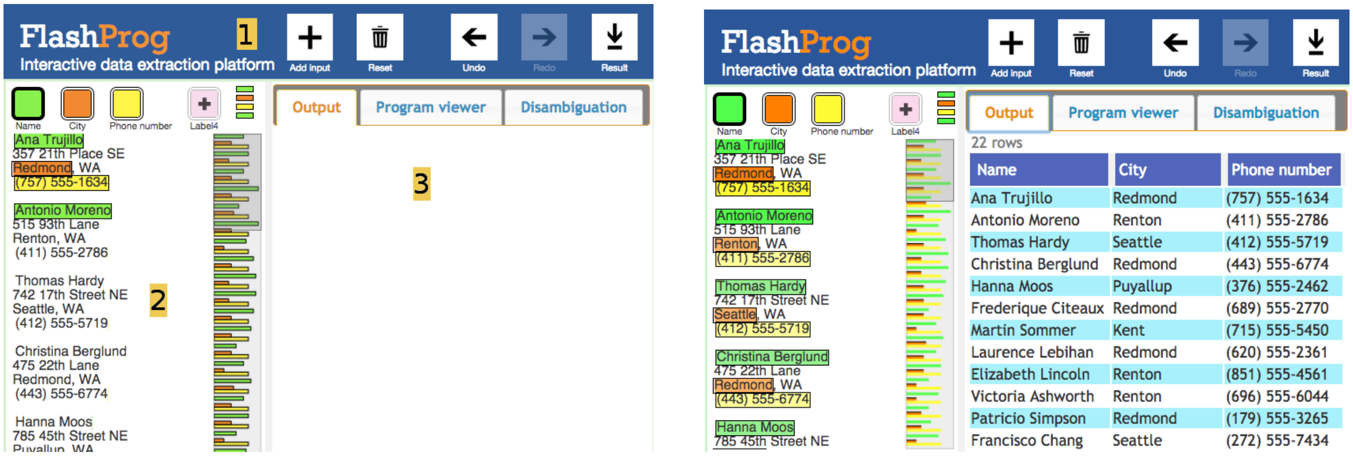
The PBE Interaction View is a tabbed pane giving users an opportunity to interact with the PBE system in three different ways: (i) exploring the produced output, (ii) exploring the learned program set paraphrased into English inside program viewer (Program Navigation), and (iii) engaging in an active learning session through the “Disambiguation” feature (Conversational Clarification).[1]

The Output pane displays the current state of data extraction result either as a relational table or as a tree. To facilitate exploration of the data, the Input Text View is scrolled to the source position of each cell when the user hovers over it. The user can also mark incorrect table rows as negative examples.

The Program viewer pane (Figure 2) lets users explore the learned programs. We concisely describe regexes that are used to match strings in the input text. For instance, “`Words/dots/hyphens∘WhiteSpace`” (the circle is an infix concatenation) represents `[-.\pLu\pLl]+o\pZs+` (viewable in code mode). To facilitate understanding of these regexes, when the user hovers over part of a regex, our UI highlights matches of that part in the text. In Figure 2, `Name-Struct` refers to the region between two consecutive names; `City-Struct` refers to the region between `City` and the end of the enclosing `Name-Struct` region. Learned programs reference these regions to extract data. For instance, `Phone` is learnt relatively to enclosing `City-struct` region: “`second line`” refers to the line in the `City` region. In addition, clicking on the triangular marker opens a list of alternative suggestions for each subexpression. We show number of highlights that will be added (or changed/removed) by the alternative program as a +number (or a -number). If the program is incorrect, the user can replace some expressions with alternatives from the suggested list (Figure 6).

The Disambiguation pane (Figure 7) presents the Conversational Clarification interaction model. The PBE engine often learns multiple programs that are consistent with the examples but produce different outputs on the rest of the document. In such cases, this difference is highlighted and user is presented with an option to choose between the two behaviors. Choosing one of the options is always equivalent to providing one more example (either positive or negative), thereby invoking the learning again on the extended specification.

[1] Note that throughout the paper, we refer to the “disambiguation” as an overall problem of selecting the program that realizes user's intent in PBE. However, in our UI we use the word “Disambiguation” as a header of a pane with one iteration of the Conversational Clarification process. We found that it describes Conversational Clarification most lucidly to the users. Hereinafter in the paper, we refer to the “Disambiguation pane” in our UI if the context does not facilitate any confusion with the “disambiguation problem”.



Ana Trujillo
357 21th Place SE
Redmond, WA
(757) 555-1634

Antonio Moreno
515 93th Lane
Renton, WA
(411) 555-2786

Thomas Hardy
742 17th Street NE
Seattle, WA
(412) 555-5719

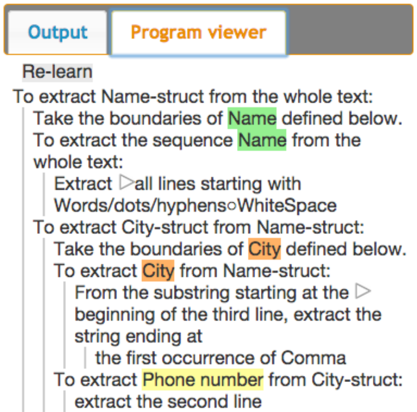
Christina Berglund
475 22th Lane
Redmond, WA
(443) 555-6774

Hanna Moos
785 45th Street NE
Puyallup, WA

22 rows

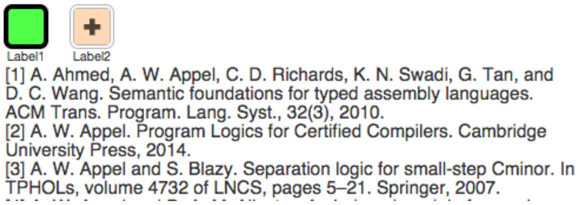
Name	City	Phone number
Ana Trujillo	Redmond	(757) 555-1634
Antonio Moreno	Renton	(411) 555-2786
Thomas Hardy	Seattle	(412) 555-5719
Christina Berglund	Redmond	(443) 555-6774
Hanna Moos	Puyallup	(376) 555-2462
Frederique Citeaux	Redmond	(689) 555-2770
Martin Sommer	Kent	(715) 555-5450
Laurence Lebihan	Redmond	(620) 555-2361
Elizabeth Lincoln	Renton	(851) 555-4561
Victoria Ashworth	Renton	(696) 555-6044
Patricio Simpson	Redmond	(179) 555-3265
Francisco Chang	Seattle	(272) 555-7434

Figure 1: FlashProg UI with PBE Interaction View in the "Output" mode, before and after the learning process. 1 – Top Toolbar, 2 – Input Text View, 3 – PBE Interaction View.



```
To extract Name-struct from the whole text:
  Take the boundaries of Name defined below.
  To extract the sequence Name from the
  whole text:
    Extract ▷all lines starting with
    Words/dots/hyphens○WhiteSpace
  To extract City-struct from Name-struct:
    Take the boundaries of City defined below.
    To extract City from Name-struct:
      From the substring starting at the ▷
      beginning of the third line, extract the
      string ending at
        the first occurrence of Comma
    To extract Phone number from City-struct:
      extract the second line
```

Figure 2: Program Viewer tab of FlashProg. It shows the extraction programs that were learned in the session in Figure 1. The programs are paraphrased in English and indented.



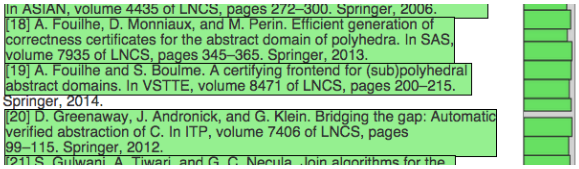
[1] A. Ahmed, A. W. Appel, C. D. Richards, K. N. Swadi, G. Tan, and D. C. Wang. Semantic foundations for typed assembly languages. ACM Trans. Program. Lang. Syst., 32(3), 2010.
[2] A. W. Appel. Program Logics for Certified Compilers. Cambridge University Press, 2014.
[3] A. W. Appel and S. Blazy. Separation logic for small-step Cminor. In TPHOLs, volume 4732 of LNCS, pages 5–21. Springer, 2007.

Figure 3: Initial input to FlashProg in our illustrative scenario: extraction of the author list from the PDF bibliography of "A Formally-Verified C Static Analyzer" [11].

Illustrative Scenario

To illustrate the different interaction models and features of FlashProg, we consider the task of extracting the set of individual authors from the Bibliography section of a paper "A Formally-Verified C Static Analyzer" [11] (Figure 3). Our model user Alice wants to extract this data to figure out who is the most cited author in papers presented at POPL 2015.

First, Alice provides an example of an outer region containing each publication record. After providing two examples, a program is learned and other publications are highlighted, but the user notices that there is an unexpected gap between two extracted regions using the bird's-eye view (Figure 4). Giving another example to also include the text "`Springer, 2014.`" fixes the problem and a correct program is learned for the publication record regions.

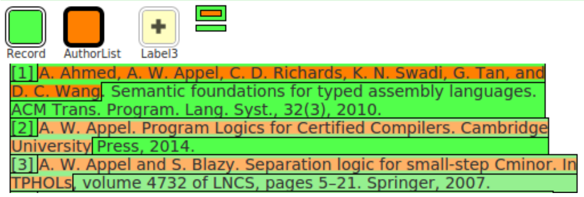


In ASIAN, volume 4435 of LNCS, pages 272–300. Springer, 2006.
[18] A. Fouilhe, D. Monniaux, and M. Perin. Efficient generation of correctness certificates for the abstract domain of polyhedra. In SAS, volume 7935 of LNCS, pages 345–365. Springer, 2013.
[19] A. Fouilhe and S. Boulmé. A certifying frontend for (sub)polyhedral abstract domains. In VSTTE, volume 8471 of LNCS, pages 200–215. Springer, 2014.
[20] D. Greenaway, J. Andronick, and G. Klein. Bridging the gap: Automatic verified abstraction of C. In ITP, volume 7406 of LNCS, pages 99–115. Springer, 2012.
[21] S. Gulwani, A. Tiwari, and G. C. Necula. Join algorithms for the

Figure 4: Bird's eye view showing discrepancy in extraction.

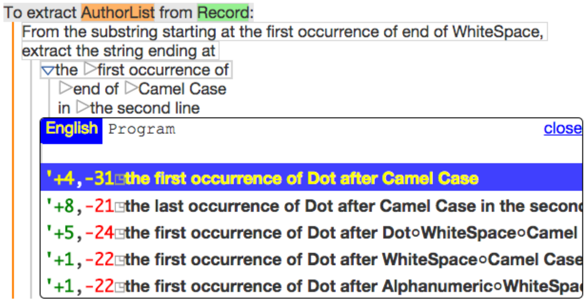
Next, Alice wants to extract the list of authors and provides an example inside the first record. After learning, she observes that the program learned is behaving incorrectly (Figure 5). At this point, Alice can provide more examples as before to fix the problem, but it is easier to switch to the Program Viewer tab, and select a correct alternative for the wrong subexpression (Figure 6). The top-ranked program for extracting the Author list from a Record is "`extract the substring starting at first occurrence of end of whitespace and ending at the first occurrence of end of Camel Case in the second line`". The sub-program for the starting position seems correct but the sub-program for the ending position seems too specific for the given example, and Alice can ask for other alternative programs that the system has learned for the end position. Hovering over each alternative previews the extraction results in the input pane. In this case, Alice hovers over the first alternative, which generates the correct result. The final learned program turns out to be "`extract everything between first whitespace and first occurrence of Dot after CamelCase`" that is correct ("Wang" is considered to be in CamelCase by FlashProg, even though it is just one word), but the logic is quite non-obvious even for a programmer to come up with.

Now Alice wants to extract each author individually, and provides two examples within the first publication record. FlashProg again does not identify all authors correctly. Alice can provide additional examples or look at the extraction pro-



[1] A. Ahmed, A. W. Appel, C. D. Richards, K. N. Swadi, G. Tan, and D. C. Wang. Semantic foundations for typed assembly languages. ACM Trans. Program. Lang. Syst., 32(3), 2010.
[2] A. W. Appel. Program Logics for Certified Compilers. Cambridge University Press, 2014.
[3] A. W. Appel and S. Blazy. Separation logic for small-step Cminor. In TPHOLs, volume 4732 of LNCS, pages 5–21. Springer, 2007.

Figure 5: An error during the author list extraction.



To extract AuthorList from Record:
From the substring starting at the first occurrence of end of WhiteSpace,
extract the string ending at
▽the ▷first occurrence of
▷end of ▷Camel Case
in ▷the second line

'+4,-31 the first occurrence of Dot after Camel Case
'+8,-21 the last occurrence of Dot after Camel Case in the secon
'+5,-24 the first occurrence of Dot after Dot∘WhiteSpace∘Camel
'+1,-22 the first occurrence of Dot after WhiteSpace∘Camel Case
'+1,-22 the first occurrence of Dot after Alphanumeric∘WhiteSpa

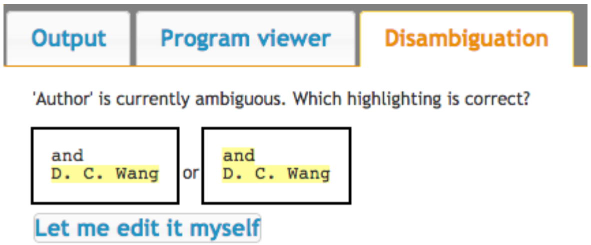
Figure 6: Program Viewer tab & alternative subexpressions.

gram, but she decides to engage the Conversational Clarification mode, and help FlashProg disambiguate between programs by answering clarifying questions (such as should the output include "`D. Richards`" or "`C. D. Richards`" and if "`and`" should be included, as shown in Figure 7). At each iteration, FlashProg asks her to choose between several possible highlightings in the unmarked portion of the document. Each choice is then communicated to the PBE system and the set of programs is re-learned. After two iterations of Conversational Clarification, FlashProg converges on a correct program, and Alice is confident in it (Figure 8).

IMPLEMENTATION

Our underlying program learning engine is a rich toolkit of generic algorithms for PBE. It allows a domain expert to easily define a *domain-specific language* (DSL) of programs that perform data manipulation tasks in a given domain [27]. The expert (DSL designer) only defines the semantics of DSL operators, from which our engine automatically generates a *synthesizer*. A synthesizer is an algorithm that, at run time, accepts a *specification* from a user, and returns a *set of DSL programs* that satisfy this specification. For instance, a specification in FlashExtract, the text processing DSL of FlashProg, is given by a sequence of positive and negative highlightings. The efficiency of our learning engine is based on two ideas from our prior work in PBE: our *synthesis algorithm* and our *program set representation*.

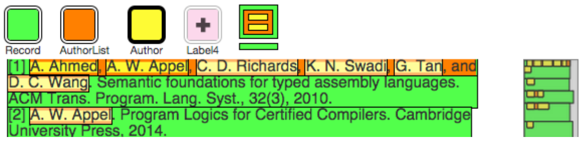
Synthesis algorithm Most prior work in PBE implement their synthesis algorithms by exhaustive search over the DSL, or delegate the task to constraint solvers [1]. In contrast, our engine employs an intelligent "top-down" search over the DSL structure, in which it iteratively transforms the examples given by an end user for the entire DSL program into the examples that should be satisfied by individual subexpressions in the program [27]. Such an approach allows FlashProg to be responsive within 1-3 seconds for each learning round, whereas state-of-the-art PBE techniques can take minutes or even hours on similar tasks. Moreover, it also allows us to generate a *set* of programs satisfying a specification, instead of a single candidate. We then use a domain-specific *ranking scheme* to select a program that will be presented to the user.



'Author' is currently ambiguous. Which highlighting is correct?

and D. C. Wang or and D. C. Wang

Figure 7: Conversational Clarification being used to disambiguate different programs that extract individual authors.



[1] A. Ahmed, A. W. Appel, C. D. Richards, K. N. Swadi, G. Tan, and D. C. Wang. Semantic foundations for typed assembly languages. ACM Trans. Program. Lang. Syst., 32(3), 2010.
[2] A. W. Appel. Program Logics for Certified Compilers. Cambridge University Press, 2014.

Figure 8: Final result of the bibliography extraction scenario.

Program set representation A typical learning session can return up to 10^{30} ambiguous programs, all consistent with the current specification [8]. Our engine makes use of a polynomial-space representation of such a program set, known as *version space algebra* (VSA). It has been introduced by Mitchell [24] in the context of machine learning, and later used by Lau et al. [15], Polozov and Gulwani [7, 27].

The key idea of VSAs is sharing of subspaces. Consider an operator $\texttt{SubStr}(s, p_1, p_2)$, which extracts a substring of s that starts at the position p_1 and ends at the position p_2. Here p_1 and p_2 can expand to various position logics, e.g. absolute ("5th character from the right") or based on regular expressions ("after the second number"). On a given example, p_1 and p_2 are known to evaluate to 1 and 4, respectively (i.e. the result of $\texttt{SubStr}(s, p_1, p_2)$ is the string $s[1 : 4]$). Importantly, both p_1 and p_2 may satisfy this specification in multiple possible ways. For example, p_1 can expand to a program "1st character from the left", or a program "$(|s| - 1)^{\text{th}}$ character from the right", or any consistent regex-based program (based on the content of s in a given example). Thus, the total number of possible consistent $\texttt{SubStr}(s, p_1, p_2)$ programs is quadratic in the number of possible consistent position programs (since any consistent p_1 can be combined with any consistent p_2).

A VSA stores these programs concisely as a *join structure* over the two program sets with learned consistent program sets for p_1 and p_2 (also represented as VSAs). Such a structure consists of the two learned program sets for p_1 and p_2 and a "join tag", which specifies that any combination of the programs sampled from these two sets is a valid combination of parameters for the `SubStr` operator. Therefore, the overall size of a VSA is typically logarithmic in the number of programs it semantically represents.

Formally, our learning engine represents program sets as a combination of shared program sets using two operators: *union* and *join*. A union of two VSAs $\tilde{N}_1$ and $\tilde{N}_2$ represents a set that is a union of two sets represented by $\tilde{N}_1$ and $\tilde{N}_2$. A join of two VSAs $\tilde{N}_1$ and $\tilde{N}_2$ represents a set that is a Cartesian product of two sets represented by $\tilde{N}_1$ and $\tilde{N}_2$. Such a representation has two major benefits: **(a)** it stores an exponential number of candidate programs using only polynomial space, and **(b)** it allows exploring the shared parts of the space of candidates, and quickly examine the alternative candidate subexpressions at any given program level.

The ideas explained above are the key to our novel Program Navigation and Conversational Clarification interaction models. We present their implementation below.

Program Navigation

The two key challenges in Program Navigation are: paraphrasing of the DSL programs in English, and providing alternative suggestions for program expressions.

Templating language

To enable paraphrasing, we implemented a high-level templating language, which maps *partial programs* into *partial English phrases*. Lau stated [14]:

"Users complained about arcane instructions such as "set the `CharWeight` *to* 1*" (make the text bold).* [...] *SMARTedit users also thought a higher-level description such as "delete all hyperlinks" would be more understandable than a series of lower level editing commands."*

Our template-based strategy for paraphrasing avoids arcane instructions by using "context-sensitive formatting rules", and avoids low-level instructions by using "idiomatic rules", solving Lau's two problems.

Paraphrasing is a conflictless bottom-up process. If possible, we use an idiom. We then remove context formatters from the template and apply them to their referenced child's template. Let us illustrate the development process with an example, a toy program named S_1:

$$\mathsf{PosPair}(\mathsf{Pos}(\mathsf{Line}(1), 1), \mathsf{Pos}(\mathsf{Line}(1), -1))$$

which evaluates to the string between the start and end of the second line. Line indexes start at 0, whereas char indexes start at 1. The relevant DSL portion is defined as a CFG:

$$\begin{array}{ll} S \coloneqq \mathsf{PosPair}(p, p) & p \coloneqq \mathsf{Pos}(L, n) \\ L \coloneqq \mathsf{Line}(n) & n \coloneqq \mathtt{int} \end{array}$$

We add three paraphrasing rules:

$$\begin{array}{rcl} \mathsf{PosPair} & \rightarrow & \text{“extract the string between \{:0\} and \{:1\}”} \\ \mathsf{Pos} & \rightarrow & \text{“the char number \{:1\} of \{:0\}”} \\ \mathsf{Line} & \rightarrow & \text{“line \{:0\}”} \end{array}$$

{:0} and {:1} refer to first and second arguments. Paraphrasing S_1 yields (parentheses added to see the paraphrase tree):

"extract the string between (the char number (1) of (line (1))) and (the char number (-1) of (line (1)))"

To differentiate the two 1, we rewrite the last two rules above with a list of dot-prefixed formatters:

$$\begin{array}{rcl} \mathsf{Pos} & \rightarrow & \text{“the \{:1.charNum\} of \{:0\}”} \\ \mathsf{Line} & \rightarrow & \text{“\{:0.lineNum\}”} \end{array}$$

charNum (resp. lineNum) is a formatter mapping ints to a char ordinal (resp. line ordinal). Formatters are lists of $\langle \text{regex}, \text{result} \rangle$ pairs modifying the template of the targeted child. Its template is then replaced by the first matching regex result. For example, the formatter for charNum (and another formatter ordinal) is:

```
charNum: [ {regex: "^1$",        result: "beginning"}, ...
           {regex: "^(\\d+)$",   result: "{:1.ordinal} char"}],
 ordinal: [ {regex: "^1$",        result: "first"},
           {regex: "^2$",        result: "second"}... ]
```

Note how we handle corner cases. Paraphrasing S_1 yields

"extract the string between (the (beginning) of (second line)) and (the (end) of (second line)))"

The paraphrasing can be made even more concise by adding *idiom rules*, which produce more natural paraphrasing for certain idiomatic expressions. An idiom rule applies to subexpressions that satisfy given equality conditions between subterms or inner terms, specified by their paths. A *path* is a colon-separated list of symbols, function names and child indexes referring to a particular node. The rule below expresses the idiom of extracting the entire line:

$$\begin{array}{c} \mathsf{PosPair}(\mathsf{Pos}(?L, 1), \mathsf{Pos}(?L, -1)) \\ \mathsf{when}\ \{\text{:0:L}\} = \{\text{:1:L}\} \end{array} \rightarrow \text{“extract the \{:L\}”}$$

S_1 is finally paraphrased into "extract the (second line)".

The limitations of this approach are mostly that all rules are written and updated manually. When the DSL changes, this is extra work. Furthermore, paraphrasing depends on order of formatters and idioms, and idiom templates may also not allow the user to explore the full program. We overcome this by letting the user switch between the paraphrase and the code (the latter being always complete).

Program alternatives

To enable alternatives, we record the original candidate program set for each subexpression in the chosen program. Since it is represented as a VSA, we can easily retrieve a subspace of alternatives for each program subexpression, and apply the domain-specific ranking scheme on them. The top 5 alternatives are then presented to the user.

Conversational Clarification

Conversational Clarification selects examples based on different outputs produced by generated programs. Each synthesis step produces a VSA of ambiguous programs that are consistent with the given examples. Conversational Clarification iteratively replaces the subexpressions of the top-ranked program with its top k alternatives from the VSA. This produces k clarification candidates (in FlashProg, k is set to 10). The clarifying question for the user is based on the *first discrepancy* between the outputs of the currently selected program P and the clarification candidateP'. Such a discrepancy can have three possible manifestations:

- The outputs of P and P' match until P selects a region r, which does not intersect any selection of P'. This leads to the question "Should r be highlighted or not?"

- The outputs of P and P' match until P' selects a region r', which does not intersect any selection of P. This leads to the question “Should r' have been highlighted?”
- The outputs of P and P' match until P selects a region r, P' selects a different region r', and r intersects r'. This leads to the question “Should r or r' be highlighted?”

For better usability (and faster convergence), we merge the three question types into one, and ask the user “What should be highlighted: r_1, r_2, or nothing?” Selecting r_1 or r_2 would mark the selected region as a positive example. Selecting “nothing” would mark both r_1 and r_2 as negative examples. After selecting an option, we convert the choice into one or more examples, and invoke a new synthesis process.

Analysis

Since Conversational Clarification is an iterative refinement of a previous synthesis process, it is guaranteed to perform several times more efficiently compared to the last process. Moreover, since we pick a clarifying question based on different outputs produced by two ambiguous candidates, *the new set of candidates is guaranteed to be smaller than the previous one*. Therefore, Conversational Clarification converges to the program(s) representing user’s intent in a finite number of rounds (if such programs exist). The number of rounds depends on the space of collisions in DSL outputs and can be exponential. In our user study and in most of our benchmarks however, the number of Conversational Clarification rounds never exceeded 5 for a single label.

A Conversational Clarification round is sound by construction (i.e. accepting a suggestion always yields a program that is consistent with both the suggestion and the prior examples). However, since our choice of clarification candidates is limited to top k alternatives at each level of the VSA, the Conversational Clarification round may be incomplete (i.e. the suggestions may not include the intended correct output). User can always provide a manual example instead of using CC suggestions in such a situation. The performance of a single Conversational Clarification round is linear in the VSA space (which is typically logarithmic in the number of ambiguous programs), since CC is implemented over our novel (recursively defined) *ranking* operation over the VSA [27].

Domain-specific languages

The generic implementation of our learning engine allows rapid development of DSLs for various data manipulation domains without the accompanying burden of designing individual synthesis algorithms or other FlashProg functionality for them. Following this methodology, we easily incorporated the following data manipulation DSLs in FlashProg:

1. FlashFill – a DSL for syntactic string transformations [7].
2. FlashExtract – a DSL for extracting textual information from semi-structured documents [17].
3. FlashRelate – a DSL for extracting relational tables from semi-structured spreadsheets [4].
4. FlashWeb – a DSL for extracting webpage content based on CSS selectors.

We design these DSLs such that they are succinct enough to enable efficient learning, yet expressive enough to support many real-world tasks. If a task can be expressed in our language, our engine will learn a program for it given sufficiently many examples. The engine fails if the language cannot express the task. For example, FlashExtract does not support arbitrary boolean conjunctions and disjunctions. Hence, if the tasks require learning a complex boolean expression, FlashExtract will not be able to perform it [7, 17, 4].

Next, we present our user study on FlashExtract below, but the functionality of FlashProg is automatically provided for any compliant DSL. We plan to incorporate more extraction domains, such as PDF documents, in future work.

EVALUATION

In this section, we present a user study to evaluate FlashProg. In particular, we address three research questions for PBE:

- RQ1: Do Program Navigation and Conversational Clarification contribute to correctness?
- RQ2: Which of Program Navigation and Conversational Clarification is perceived more useful for data extraction?
- RQ3: Do FlashProg’s novel interaction models help alleviate typical distrust in PBE systems?

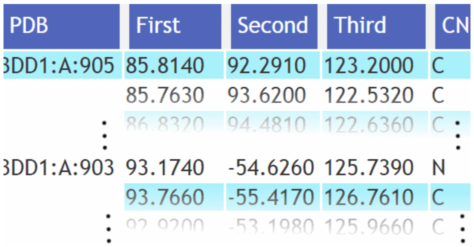
User study design

Because our tasks can be solved without any programming skills, we performed a within-subject study over an heterogeneous population of 29 people: 4 women aged between 19 and 24 and 25 men aged between 19 and 34. Their programming experience ranged from none (a 32-year man doing extraction tasks several times a month), less than 5 years (8 people), less than 10 (9), less than 15 (8) to less than 20 (3). They reported performing data extraction tasks never (4 people), several times a year (7), several times a month (11), several times a week (3) up to every day (2).

We selected 3 files containing several ambiguities these users have to find out and to resolve. We chose these files among anonymized files provided by our customers. Our choice was also motivated by repetitive tasks, where extraction programs are meant to be reused on other similar files. The three files are the following:

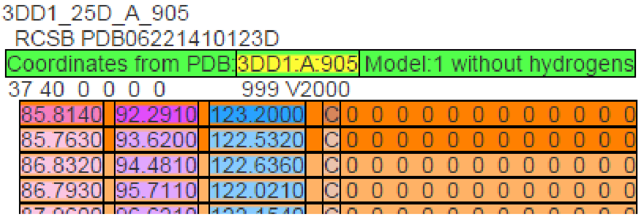
1. **Bank listing.** List of bank addresses and capital grouped by state. The postal code can be ambiguous.
2. **Amazon research.** The text of the search results on Amazon for the query “chair”. The data is visually structured as a list of records, but contains spacing and noise.
3. **Bioinformatic log.** A log of numerical values obtained from five experiments, from bioinformatics research (Figure 10). Straightforward extraction misses one experiment.

We first provided users a brief video tutorial using the address file as example (Figure 1, youtu.be/JFRI4wIR0LE). The video shows how to perform two extractions and to use features such as undo/redo. It partially covers the Program Viewer tab and the Disambiguation tab. It explains that these features will or will not be available, depending on the tasks. When users start FlashProg, they are given the same file as in the video. A pop-up encourages them to play with it, and



PDB	First	Second	Third	CN
3DD1:A:905	85.8140	92.2910	123.2000	C
	85.7630	93.6200	122.5320	C
⋮	86.8320	94.4810	122.6360	C ⋮
3DD1:A:903	93.1740	-54.6260	125.7390	N
	93.7660	-55.4170	126.7610	C
⋮	92.9200	-53.1980	125.9660	C ⋮

Figure 9: Bioinformatic log: Result sample.



```
3DD1_25D_A_905
 RCSB PDB06221410123D
Coordinates from PDB:3DD1:A:905 Model:1 without hydrogens
37 40  0  0  0  0            999 V2000
  85.8140   92.2910  123.2000   C 0  0  0  0  0  0  0  0  0  0  0  0
  85.7630   93.6200  122.5320   C 0  0  0  0  0  0  0  0  0  0  0  0
  86.8320   94.4810  122.6360   C 0  0  0  0  0  0  0  0  0  0  0  0
  86.7930   95.7110  122.0210   C 0  0  0  0  0  0  0  0  0  0  0  0
```

Figure 10: Highlighting for obtaining Figure 9.

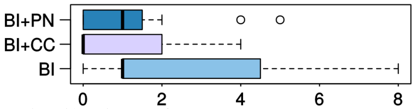


Figure 11: Distribution of error count across environments. Both Conversational Clarification (CC) and Program Navigation (PN) significantly decrease the number of errors.

to continue when they feel ready. The Program Viewer tab and the Disambiguation tab are both available at this point.

We then ask users to perform extraction on the three files. For each extraction task, we provide a result sample (Figure 9). Users then manipulate FlashProg to generate the entire output table corresponding to that task. We further instruct them that the order of labels do not matter, but they have to rename them to match our result sample.

To answer RQ1, we select a number of representative values across all fields for each task, and we automatically measure how many of them were incorrectly highlighted. These values were selected by running FlashProg sessions in advance ourselves and observing insightful checkpoints that require attention. In total, we selected 6 values for task #1, 13 for task #2 and 12 for task #3. We do not notify users about their errors. This metric has more meaning than if we recorded all errors. As an illustration, a raw error measurement in the third task for a user forgetting about the third main record would yield more than 140 errors. Our approach returns 2 errors, one for the missing record, and one for another ambiguity that needed to be checked but could not. This makes error measurement comparable across tasks.

Environments To measure the impact of Program Navigation and Conversational Clarification interaction models independently, we set up three interface environments.

Basic Interface (BI). This environment enables only the Colored Data Highlighting interaction model. It includes the following UI features: the labeling interface for mouse-triggered highlighting, the label menu to rename labels, to switch between them and the Output tab.

BI + Program Navigation (BI + PN). Besides the Colored Data Highlighting, this interface enables the Program Navigation interaction model, which includes the Program Viewer tab and its features (e.g. Regular expression highlighting, Alternative subexpression viewer).

BI + Conversational Clarification (BI + CC). Besides the Colored Data Highlighting, this environment enables the Conversational Clarification interaction model, which includes the Disambiguation tab.

To emphasize PN and CC, the system automatically opens the matching tab, if they are part of the environment.

Configurations To compensate the learning curve effects when comparing the usefulness of various interaction models, we set up the environments in three configurations A, B, and C. Each configuration has the same order of files/tasks, but we chose three environment permutations. As we could not force people to finish the study, the number of users per environment is not perfectly balanced.

	Tasks			
Config.	1. Bank	2. Amazon	3. Bio log	# of users
A	BI + PN	BI + CC	BI	8
B	BI	BI + PN	BI + CC	12
C	BI + CC	BI	BI + PN	9

Survey To answer RQ2 and RQ3, we asked the participants about the perceived usefulness of our novel interaction models, and the confidence about the extraction of each file, using a Likert scale from 1 to 7, 1 being the least useful/confident.

Results

We analyzed the data both from the logs collected by the UI instrumentation, and from the initial and final surveys. If a feature was activated, we counted the user for statistics even if he reported not using it.

RQ1: Do Program Navigation and Conversational Clarification contribute to correctness? **Yes. We have found significant reduction of number of errors with each of these new interaction models** (See Figure 11). Our new interaction models reduce the error rate in data extraction without any negative effect on the users' extraction speed. To obtain this result, we applied the Wilcoxon rank-sum test on the instrumentation data. More precisely, users in BI + CC ($W = 78.5$, $p = 0.01$) and BI + PN ($W = 99.5$, $p = 0.06$) performed better than BI, with no significant difference between the two of them ($W = 94$, *n.s.*). There was also no statistically significant difference between the completion time in BI and completion time in BI + CC ($W = 178.5$, *n.s.*) or BI + PN ($W = 173$, *n.s.*).

RQ2: Which of Program Navigation and Conversational Clarification is perceived more useful for data extraction? **Conversational Clarification is perceived more useful than Program Navigation** (see Figure 12a and Figure 12b). Comparing the user-reported usefulness between the Conversational Clarification and the Program Navigation, on a scale from 1 (not useful at all) to 7 (extremely useful), the Con-

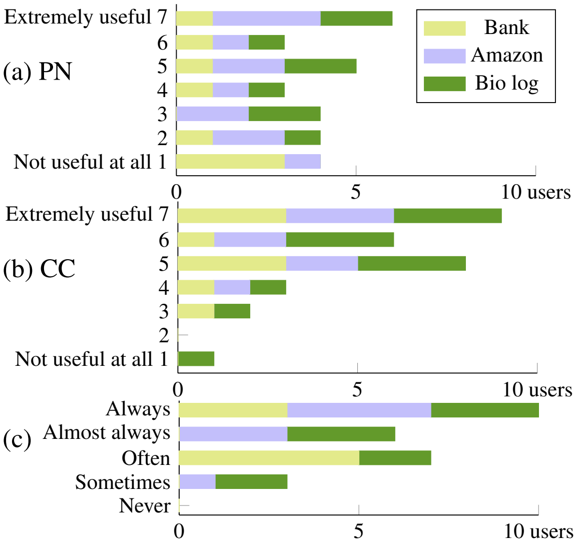


Figure 12: User-reported: **(a)** usefulness of PN, **(b)** usefulness of CC, **(c)** correctness of one of the choices of CC.

versational Clarification has a mean score of 5.4 ($\sigma = 1.50$) whereas the Program Navigation has 4.2 ($\sigma = 2.12$) Only 4 users out of 29 score Program Navigation more useful than Conversational Clarification, whereas Conversational Clarification is scored more useful by 15 users.

RQ3: Do FlashProg*'s novel interaction models help alleviate typical distrust in PBE systems?* **Yes for Conversational Clarification.** Considering the confidence in the final result of each task, tasks finished with Conversational Clarification obtained a higher confidence score compared to those without ($W = 181.5$, $p = 0.07$). No significant difference was found for Program Navigation ($W = 152.5$, *n.s.*). Regarding the trust our users would have if they had to run the learned program on other inputs, we did not find any significant differences for Conversational Clarification ($W = 146$, *n.s.*) and Program Navigation ($W = 161$, *n.s.*) over only BI.

Regarding the question "How often would you use FlashProg, compared to other extraction tools?", on a Likert scale from 1 (never) to 5 (always), 4 users answered 5, 17 answered 4, 3 answered 3, and the remaining 4 answered 2 or less. Furthermore, all would recommend FlashProg to others. When asked how excited would they be to have such a tool on a scale from 1 to 5, 8 users answered 5, and 15 answered 4.

The users' trust is supported by data: Perceived correctness is negatively correlated with number of errors (Spearman $\rho = -0.25, p = 0.07$). However, there is no significant correlation between number of errors made and the programming experience mapped between 0 and 4 ($\rho = -0.09$, *n.s.*).

Other results. We observed that only 13 (45%) of our users used the Program Viewer tab when it was available. These 13 users having experienced Program Navigation got mixed feelings about it. A 22-year woman with more than 5 years of programming experience gave a positive review: "I absolutely loved [regular expression highlighting]. I think that perfectly helps one understand what the computer is thinking at the moment and to identify things that were misunderstood". According to a 27-year man with more than 10 years of programming experience, the interaction was not easy enough: "the program [is] quite understandable but it was not clear how to modify the program". 9 users out of 13 did not report using the Alternative subexpression viewer when using the Program Navigation.

On the other hand, 27 (93%) used the Disambiguation tab when it was available. Users appreciated it. The previous woman said: "in the last example, in which I didn't have [Conversational Clarification] as an option, I felt like I miss it so much". A 27-year man with 5+ years of programming experience said: "It always helps me to find the right matching". A 19-year old novice programmer woman said: "The purpose of each box wasn't clear enough, but after the text on left became highlighted (hovering the boxes), the task became easier". Although there were tooltips, some users were initially confused about how we presented negative choices with XXX crossing the answer.

Discussion

With so many experienced users, we did not expect that only half of them would interact with Program Navigation, and even less with the Alternative subexpression viewer. To ensure usability, we developed FlashProg and Program Navigation iteratively based on the feedback of many demo sessions and a small 3-user pilot study before running the full user study. We did not receive any specific complaints about the paraphrasing itself, although it certainly required substantial time to understand their semantics. In the tasked they solved, users might then have thought that it would take more time to figure out where the program failed, and to find a correct alternative, than to add one more example. We believe that in other more complex scenarios, such as with larger files or multiple files, the time spent using Program Navigation could be perceived as more valuable and measured as such. The decrease of errors may then be explained by the fact that when Program Navigation was turned on, users have stared at FlashProg more and took more time to catch errors.

The negative correlation between the confidence of users in the result and the number of errors is insightful. Although we asked them to make sure the extraction is correct and never told them they did errors, users making more errors (thus unseen) reported to be less sure about the extraction. The problem is therefore not just about alleviating the users' typical distrust in the result, it is really about its correctness.

We also acknowledge that several factors may be a limitation of this study: (a) we have a limited amount of heterogeneous users; (b) the time was uncontrolled, thus we could not prevent users from getting tired or from pausing in the middle of extraction tasks; (c) besides the 29 users having completed all the study, more than 50 users who decided to start the study stopped before finishing the last task (this explains the unbalanced number of users for each condition). Thus, they were not part of the qualitative correlations (e.g. between confidence and errors), but we did include each finished task for the error metrics; (d) if a user extracts all regions manually, replacing a record not covered by the checkpoints by another, we do not measure this error (false negatives).

CONCLUSION AND FURTHER WORK

We have implemented FlashProg, a prototype PBE system for data extraction and manipulation that supports two novel user interaction models for disambiguation in PBE, namely Program Navigation and Conversational Clarification. In user studies, where users were given three data extraction tasks, we found that a significant majority of users found the tool effective and were confident in the results. This confidence is supported by data: both models significantly reduced the number of extraction errors. Further, the users found the proactive behavior of Conversational Clarification very useful, and preferred it to the Program Navigation interface.

As PBE technologies such as FlashProg are made more widely available in the marketplace, we will better understand the interplay between the user's task understanding and the tool's ability to support them. Beyond our current work, there are many opportunities for improvements, including helping users with greater automation, helping them deal with incomplete and sometimes incorrect data, and identifying when the user has made a mistake in their examples.

ACKNOWLEDGMENTS

This work was sponsored by Microsoft Research. Mikael, Gustavo, Vu, and Alex were supported by a one-year position, and Maxim was supported by a six-month position at MSR.

REFERENCES

1. Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M K Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *FMCAD*. IEEE.
2. Saleema Amershi, James Fogarty, Ashish Kapoor, and Desney S. Tan. 2009. Overview based example selection in end user interactive concept learning. In *UIST*. 247–256.
3. Saleema Amershi, James Fogarty, Ashish Kapoor, and Desney S. Tan. 2011. Effective End-User Interaction with Machine Learning. In *AAAI*.
4. Dan Barowy, Sumit Gulwani, Ted Hart, and Ben Zorn. 2015. FlashRelate: Extracting Relational Data from Semi-Structured Spreadsheets Using Examples. In *PLDI*.
5. Allen Cypher (Ed.). 1993. *Watch What I Do: Programming by Demonstration*. MIT Press.
6. John Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. In *PLDI*.
7. Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *POPL*. 317–330.
8. Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *ACM* 55, 8 (2012).
9. Sumit Gulwani, Jose Hernandez-Orallo, Emanuel Kitzelmann, Stephen Muggleton, Ute Schmid, and Ben Zorn. 2015. Inductive Programming Meets the Real World. *To appear in Commun. ACM* (2015).
10. Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *ICSE*. 215–224.
11. Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A Formally-Verified C Static Analyzer. In *POPL*. 247–259.
12. Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive visual specification of data transformation scripts. In *SIGCHI*. ACM, 3363–3372.
13. Jürgen Landauer and Masahito Hirakawa. 1995. Visual AWK: a model for text processing by demonstration. In *IEEE Symposium on Visual Languages*. 267–267.
14. Tessa Lau. 2009. Why Programming-By-Demonstration Systems Fail: Lessons Learned for Usable AI. *AI Magazine* 30, 4 (2009), 65–67.
15. Tessa Lau, Pedro Domingos, and Daniel S Weld. 2000. Version Space Algebra and its Application to Programming by Demonstration. In *ICML*. 527–534.
16. Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. 2001. Learning repetitive text-editing procedures with SMARTedit. *Your Wish Is My Command: Giving Users the Power to Instruct Their Software* (2001), 209–226.
17. Vu Le and Sumit Gulwani. 2014. FlashExtract: a framework for data extraction by examples. In *PLDI*. 542–553.
18. Alan Leung, John Sarracino, and Sorin Lerner. 2015. Interactive Parser Synthesis by Example. In *PLDI*.
19. H. Lieberman. 2001. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann.
20. Richard G. McDaniel and Brad A. Myers. 1999. Getting More Out of Programming-by-Demonstration. In *CHI*. 442–449.
21. Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler W. Lampson, and Adam Kalai. 2013. A Machine Learning Framework for Programming by Example. In *ICML*.
22. Robert C. Miller and Brad A. Myers. 2001. Outlier finding: focusing user attention on possible errors. In *UIST*. 81–90.
23. Robert C. Miller and Brad A. Myers. 2002. LAPIS: Smart editing with text structure. In *CHI '02*. ACM, 496–497.
24. Tom M Mitchell. 1982. Generalization as search. *Artificial intelligence* 18, 2 (1982), 203–226.
25. Brad A. Myers and William Buxton. 1986. Creating highly-interactive and graphical user interfaces by demonstration. In *SIGGRAPH*. 249–258.
26. Brad A. Myers, Richard G. McDaniel, and David S. Kosbie. 1993. Marquise: creating complete user interfaces by demonstration. In *INTERACT*. 293–300.
27. Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *OOPSLA*.
28. Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2014. Programming by Example Using Least General Generalizations. In *AAAI*. 283–290.
29. Christopher Scaffidi, Brad A. Myers, and Mary Shaw. 2008. Topes: reusable abstractions for validating data. In *ICSE*. 1–10.
30. Yanyan Shen, Kaushik Chakrabarti, Surajit Chaudhuri, Bolin Ding, and Lev Novik. 2014. Discovering queries based on example tuples. In *SIGMOD*. 493–504.
31. Ian H. Witten and Dan Mo. 1993. TELS: Learning Text Editing Tasks from Examples. In *Watch What I Do*, Allen Cypher (Ed.). MIT Press, 183–203.
32. Kuat Yessenov, Shubham Tulsiani, Aditya Krishna Menon, Robert C. Miller, Sumit Gulwani, Butler W. Lampson, and Adam Kalai. 2013. A colorful approach to text processing by example. In *UIST*. 495–504.