Precision Interfaces

Haoci Zhang
zhanghaoci@gmail.com
Tsinghua University

Thibault Sellam
tsellam@cs.columbia.edu
Columbia University

Eugene Wu
ewu@cs.columbia.edu
Columbia University

ABSTRACT

Building interactive tools to support data analysis is hard because it is not always clear what to build and how to build it. To address this problem, we present Precision Interfaces, a semi-automatic system to generate task-specific data analytics interfaces. Precision Interface can turn a log of executed programs into an interface, by identifying micro-variations between the programs and mapping them to interface components. This paper focuses on SQL query logs, but we can generalize the approach to other languages. Our system operates in two steps: it first build an interaction graph, which describes how the queries can be transformed into each other. Then, it finds a set of UI components that covers a maximal number of transformations. To restrict the domain of changes to be detected, our system uses a domain-specific language, PILang. We give a full description of Precision Interface's components, showcase an early prototype on real program logs and discuss future research opportunities.

1 INTRODUCTION

Data analysis and exploration tools let users navigate their datasets through interface components such as dropdown lists, sliders, or buttons. Those applications dramatically accelerate analysis by abstracting out a *common* set of operations and lifting them into the visual domain [18]. A successful application of this principle is Tableau, which optimizes OLAP exploration [22]. Another example is Crossfilter, which targets filtering [21]. Yet, building those applications is hard. The process involves high *development costs* in terms of time or expertise, combined with the reality that it is not always clear *what to build*. Therefore, interfaces do not exist for all but the most common and highest profile analysis tasks.

One approach is to provide tools and libraries that make it easier, perhaps for even end-users, to build interfaces. This is the rationale behind Shiny, a framework that helps statisticians quickly create Web interfaces for R scripts. Similarly, tools such as Sikuli [25] or Microsoft Access enable users with no engineering background to build software. Although easier to use than lower level libraries such as NodeJS or Bootstrap, they still require learning and practice. To illustrate, Shiny's Website claims that "no HTML, CSS or JavaScript knowledge [is] required", but its users need to understand reactive programming. Simply put, programming is hard [5].

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 the author(s) 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.
HILDA'17, Chicago, IL, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
978-1-4503-5029-7/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3077257.3077261

```
Q1: SELECT * FROM Sales WHERE Country = 'US'
Q2: SELECT * FROM Sales WHERE Country = 'UK'

Q3: SELECT TOP 5 * FROM Sales
Q4: SELECT * FROM Sales
```

(a) Two pairs of consecutive queries.

(b) Matching interface.

Figure 1: Example of automated interface design.

When task-specific interfaces are not available, users default to more generic systems. For example, Tableau is a powerful interface for performing OLAP-based exploration, however any given task only utilizes a small fraction of the interface's capabilities (Section 5 describes a case study in more detail). In practice, users will use whatever tools are available on-hand, or rely on technical experts to perform the analysis on their behalf. This approach has two important limits. It is not *discoverable* [20]: users often struggle to identify the features that will let them perform their analysis. It is not *efficient*: the short cuts that could be tailored to a common task are do not exist in more general systems. Ideally, users should have interfaces tailored to their set of tasks [6, 10, 15, 23, 26]. The programming languages community has seen this pattern in the rise of *domain specific languages* [7] and this can be viewed as the analogy for visual interfaces.

To this end we argue for *Precision Interfaces*, an automatic tool to generate task-specific data analytics interfaces. We believe that two observations point towards the promise of Precision Interfaces. First, modern applications [1, 2] and analysis frameworks [8] are increasing storing rich metadata about user analysis operations, including the programs that are run; simply consider the query logs that nearly all databases maintain. These traces indirectly capture the user's analytic needs and may be mined to identify patterns and common analyses that can be translated into interfaces. Second, data analysis is inherently incremental [3]. Consequently, the programs in the log also change incrementally. By identifying these incremental changes, we can more readily map them to interface components. With the confluence of these observations, we hope to move towards a future where *"no interface is left behind"*.

In the rest of this paper, we will describe how to build data analytics interfaces from program logs. Our current prototype and examples focus on *SQL query logs*, however the techniques can apply to any other language. The main idea is to detect small differences between programs, and map those to user interactions. Consider

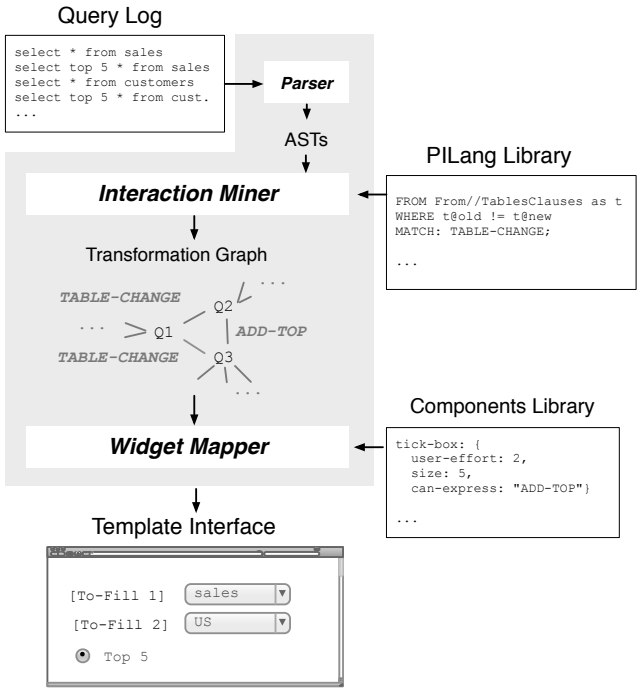

Figure 2: Overview of PI's architecture.

for instance the two pairs of SQL queries pictured in Figure 1a. We can transform Q1 into Q2 by changing the equality predicate in the `WHERE` clause. Similarly we can obtain Q4 from Q3 by adding a `TOP 5` statement. Precision Interfaces can recognize those interaction patterns, and infer an interface from them as shown in Figure 1b.

The rest of this paper is organized as follows. In Section 2, we give an overview of the PI pipeline. Sections 3 and 4 focus on two specific aspects of our system: how to describe interactions, and how to map widgets to interactions. We present a use case with real data in Section 5. We conclude in Section 6.

2 SYSTEM OVERVIEW

Precision Interfaces generates tailored interfaces from sequences of queries expressed in a query log. It does so by mapping structural changes between the queries (e.g., adding an attribute to the `SELECT` clause) to user interactions in a generated web application (e.g., clicking a button, dragging a label). As illustrated in Figure 2, Precision Interfaces employs a two step process. First, the *Interaction Miner* transforms the query log into a *transformation graph* where each query is a node and edges represent simple structural changes between the queries. Second, the *Interface Generator* maps the transformation graph to interactions in an application interface.

Detecting interactions in general is very challenging because differences between two queries may be arbitrarily complex. Our main insight is that the set of commonly used UI components and interactions—such as form elements, selection boxes, hovering, clicking—have a limited expressiveness. This observation simplifies the types of query differences that the system must consider. The rest of this section describes the key design consideration for each system component.

Parsing. Although this paper focuses on precision interfaces for SQL query logs, our vision is to build a general system that can be applied to different programming languages. To this end, program strings are not the appropriate abstraction because they lack the necessary structure and semantics for detecting structural program changes. Instead, we assume the existence of a language grammar and a parser that maps the program log into a sequence of abstract syntax trees (ASTs).

Interaction Miner. This component runs tree matching algorithms between pairs of ASTs to identify sub-tree differences. One major challenge is to identify the types of differences that can be mapped to user interactions. For instance, mapping two completely different queries such as Q1 and Q3 in Figure 1a may not help us build a practical interfaces. In our current implementation, developers use our PILang language to pre-specify the set of desirable tree differences and add them to the *Interactions Library* (Section 3). For instance, one PILang statement may be "only a number in the `WHERE` clause changed", or "an expression was added to the `SELECT` clause". If a statement matches a pair of ASTs, the Interaction Miner creates an edge in the transformation graph between the two corresponding queries that is typed by the PILang statement. Therefore, running tree matching between all pairs of queries in the query log will produce a transformation graph. In a near future, we will study automated methods to map interactions and widget specifications.

Interface Generator. We take a two step approach towards generating interfaces. First we map edges in the transformation graph to abstract UI components (e.g., radio buttons, slider, hover) in the interface. This task is a challenge because we must take into consideration the resulting interface's *coverage*—meaning the set of queries that the interface can express—as well as its *complexity*—meaning the difficulty for users to understand the interface. For instance, a trivial interface would simply map each query in the log to a button that executes the query and presents the results. Such a UI would have a high coverage but also high complexity. Similarly, edges that represent a numerical value change (e.g., changing a threshold in the `WHERE` clause) could be represented as a slider, a set of radio buttons, or a textbox. Each options has its own trade-offs, depending on the range of options that the widget must cover, how much complexity the interface allows and how frequently the component is accessed. Section 4 describes our approach for selecting UI components and balancing these trade-offs.

Once we have selected a set of abstract UI components, the second challenge is to populate the components with data (e.g., specify the minimum and maximum values of a slider), lay out the components, and render the interface. Our current implementation simply renders the components is a grid-based web template and allows the user to populate, customize and reposition them. We are working on automating this step.

Discussion. Our graph-based formulation provides considerable flexibility in the types of interfaces that we can generate by simply changing the subset of query log that Precision Interfaces analyzes. For instance, we might generate a fully expressive but complex interface by considering the complete query log. In contrast, partitioning the log by analyst generates analyst-specific interfaces to

```
// PI_Lang statement 1:
FROM project//projectclause AS cols
WHERE cols@old subset cols@new AND |cols@old| = |cols@new|+1
MATCH ???

// Example of matched queries for statement 1:
SELECT region, revenue FROM clients
SELECT revenue FROM clients

// PI_Lang statement 2:
FROM from//tableclause//tablename as T
WHERE T@old not equal T@new AND |T| = 1
MATCH ???

// Example of matched queries for statement 2:
SELECT * FROM Clients
SELECT * FROM Regions
```

Figure 3: Examples of PI-Lang statements and matched pairs of queries.

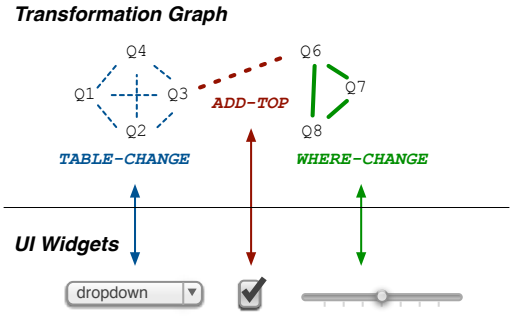

Figure 4: Examples of mapping the transformation graph to UI components.

each analyst, while partitioning by task can generate task-specific interfaces.

3 PILANG

We now describe *PILang*, a domain-specific language to express structural differences between two ASTs T_1 and T_2.

A PILang statement is composed of three clauses. The `FROM` clause specifies where differences occur. It binds range variables to paths in the ASTs, using an XPath-like syntax. The semantics is that T_1 and T_2 are identical except for the sub-trees rooted at the matching nodes.

The `WHERE` clause is a boolean expression over the range variables that specifies how they may differ. The statement generates a match when the expression evaluates to true. The suffixes `@new` and `@old` can be appended to a range variable to reference the corresponding nodes in T_1 and T_2, respectively. Finally, we support convenience expressions to help perform set comparisons between the two versions of the path expressions. For instance, `T@old subset T@new` specifies that new nodes were inserted into `T`, whereas `|T| = 1` checks that there is only one matching node.

The `MATCH` clause labels the statement. In our implementation, we model the range variables as relational tables and translate PILang into SQL. In the future, it can also expose the range variables that have changed so that their values can be dynamically bound to UI component state.

To illustrate, the following statement identifies pairs of queries with different string literals in an equality expression within their `WHERE` clause (Figure 1a):

```
FROM where//expr[op="="]//strliteral AS T
WHERE T@old not equal T@new AND |T| = 1
MATCH change_where_equal
```

The `FROM` clause matches all string literal nodes that are children of equality expressions in the filter clause. These nodes are bound to `T`. The `WHERE` clause checks that there is a single string literal that has changed (and implicitly that nothing else in the ASTs have changed). If there is a match, then we add an edge between the two input ASTs and label the edge `change_where_equal`.

Figure 3 presents two additional examples of PI-Lang statements, along with matching pairs of queries. Note that PILang statements are language agnostic and can be expressed over any programs that can be parsed into ASTs. Currently, we are developing a library of standard transformations for SQL and plan to extend support for both query languages such as SPARQL and HIVE, as well as programming languages such as R and Python.

4 INTERFACE GENERATION

We model the interface generation problem as identifying a mapping from sets of edges in the transformation graph to UI components that can express those edges. For instance, Figure 4 shows how edges that describe change the table in the `FROM` clause of a query may be mapped to a dropdown to select from the set of tables in the database; adding a `TOP 5` clause may map to a check box, whereas changes to a numerical attribute may map to a textbox or a slider.

In general, there can be many possible mappings to generate interfaces, and the natural question is "what is a good interface?". Interface theory literature has decomposed the data analysis process into high level steps and identified the sources of friction that can impede user progress [11, 16]. These sources include mapping high level goals to interface operations—which is impeded by complex interfaces—and fatigue from physically performing the operations. Based on this theory, our optimization follows three principles: *coverage*, *simplicity*, and *efficiency*.

The interface should maximize *coverage* in terms of the proportion of the graph that the interface can express. Trivially, an interface can achieve full coverage by mapping each program to a button that executes the corresponding program when pressed. However, such an interface will have high complexity and it will be challenging for a user to identify the appropriate button to click. For this reason, we emphasize interface *simplicity* by reducing the set of interaction components that are used in the interface. However, a large query input box has full coverage and is simple, but defeats the original purpose of designing an interactive interface. Thus, we seek to maximize *efficiency*, which is modeled as the amount of human effort needed to express any given analysis.

Given a transformation graph $(\mathcal{V}, \mathcal{E})$ with nodes $\mathcal{V}$ and edges $\mathcal{E}$, a mapping $M = \{(E_i, i_i) | E_i \subseteq \mathcal{E}, i_i \in \mathcal{I}\}$ maps a subset of edges E_i to an interface component i_i selected from a pre-defined interaction library $\mathcal{I}$. The overall problem statement is:

```
SELECT "ontime"."distance" AS "distance",
    SUM("ontime"."arrdelay") AS "sum:arrdelay:ok",
    SUM("ontime"."depdelay") AS "sum:depdelay:ok"
FROM "public"."ontime" "ontime"
GROUP BY 1
HAVING (MIN("ontime"."distance") >= 30.99)
   AND (MIN("ontime"."distance") <= 4983.00))

SELECT "ontime"."distance" AS "distance",
    SUM("ontime"."arrdelay") AS "sum:arrdelay:ok",
    SUM("ontime"."depdelay") AS "sum:depdelay:ok"
FROM "public"."ontime" "ontime"
GROUP BY 1
HAVING (MIN("ontime"."distance")  >= 30.99)
   AND (MIN("ontime"."distance") <= 2863.00))
```

Figure 5: Pair of queries from Tableau's logs, with a value change in the `WHERE` clause.

Definition 4.1 (Component Mapping). Given a transformation graph $(\mathcal{V}, \mathcal{E})$, identify the optimal mapping

$$M^* = \text{argmin}_M \, C_e(M) \quad s.t. \;\; C_c(M) < S_{max} \tag{1}$$

We seek to minimize the interaction cost $C_e(M)$ to transform any query from the log to any other, subject to a constraint on the interface complexity $C_c(M)$.

$C_e(M)$ is the average cost to transform between all the queries in the log q_i and q_j. We assume that it costs $c_e(i_i)$ to traverse an edge in the graph by using a given interaction $i_i \in M$. Thus, the cost to transform between the two queries $c_e(q_i, q_j; M)$ is the minimum cost path that only uses interactions in M. If such a path doesn't exist, then we assign a default cost *penalty*. With those notations, we can express $C_e(M)$ as the average cost between all query pairs in the log:

$$C_e(\mathcal{L}, M) = \frac{1}{|\mathcal{L}|} \sum_{q_i, q_j \in \mathcal{L}^2} c_e(q_i, q_j; M) \tag{2}$$

Our prototype simply considers all adjacent pairs of queries in the log.

We approximate the interface complexity by assigning each UI component a complexity score $c_c(i)$, and model the total interface complexity as the sum of all components:

$$C_c(M) = \sum_{(e,i) \in M} c_c(i)$$

In future versions, we intend to use complexity measures from the interface literature [17, 19].

Solution Sketch. The problem described in Definition 4.1 is NP-hard, as it is a generalization of the knapsack problem. We approximate the solution with a greedy heuristic. At each step, PI computes all the possible widget-transformation assignments, eliminates those that violate the complexity constraint, and choses the one which leads to the best improvement of the objective function $C_e(M)$. The system then removes all the edges and vertices concerned with the corresponding transformation, and reiterates the procedure on the reduced graph. The algorithm stops when there is no space left on the interface, that is, when $C_c(M) \geq S_{max}$.

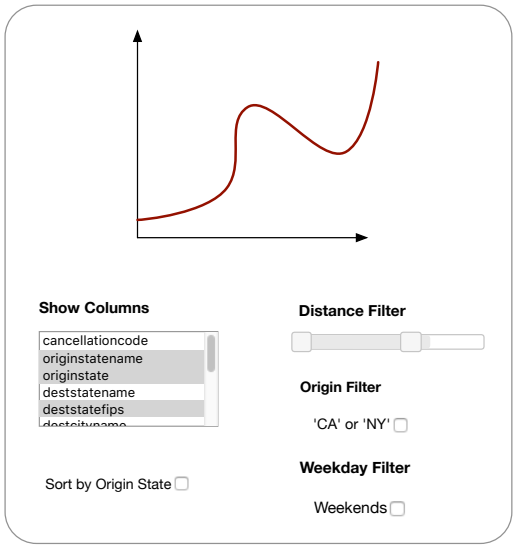

Figure 6: Interfaces generated by Precision Interfaces for the first student, with a mock-up output. We filled the data in the components, placed them on the page and wrote captions through the template generated by our system.

5 EARLY RESULTS

We now present experiments with on our prototype implementation. We asked Computer Science students to analyze the On-Time Database[1] with Tableau and collected the generated SQL queries. Our aim is to show that (1) each user only uses a small set of analysis operations, (2) Precision Interfaces can recognize those patterns from the query logs and (3) Precision Interfaces can automatically produce custom interfaces for each user.

Setup. We asked students to answer 3 out of 12 predetermined questions (e.g., "how delayed are flights to California") and answer one free-form question ("tell us something you find interesting"). We report an analysis based on the two longest query logs we collected (from two different students), which contain 167 and 137 queries respectively. We used 9 PILang statements and the default generation parameters for both logs, and simply report our results.

Results. Figure 6 demonstrates the first interface generated by our system, along with mock-up outputs[2]. Our first student decided to analyze the cause of flight delays by projecting and selecting subsets of the OnTime dataset. The interface presented Figure 6 expresses 166 out of the 167 queries that she produced, using only 5 components.

In this interface, the main component is the "Show Columns" list-box on the top left, which lets the student select which columns of the table to visualize. The tick box at the bottom toggles sorting by State. The right part of the interface consists of three filters. The top filter restricts the flight distance using a range slider. The second one toggles whether or not to filter the flights from either

[1] 521,000 rows and 91 columns. `https://www.transtats.bts.gov`

[2] This paper focuses on UI inputs. See related work [9, 13, 24] for automatic visualization generation.

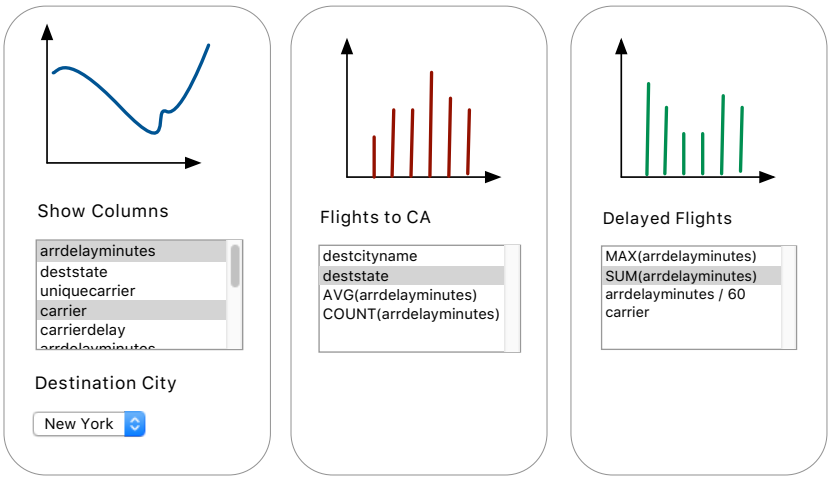

Figure 7: Interfaces generated by Precision Interfaces for the second student, with mock-up outputs.

New York or California. The bottom filter restricts the analysis to weekend flights.

Figure 7 represents the UI generated for the second student. This interface is more complex because the user performed three distinct subtasks. The leftmost panel lets her analyze all the flights in the database. The central panel focuses on flights to California. The rightmost panel focuses on delayed flights. The interface covers 120 out of 137 queries in the log.

To understand why Precision Interfaces chose to generate three separate interfaces, we plot the transformation graph in Figure 8. Recall that each edge corresponds to a transformation between two queries—for instance, the blue edges represent changes in the `SELECT` clause. Thus each isolated cluster represents a distinct set of analyses, either by focusing on a different subset of the database, or by executing structurally different queries. If the user had performed incremental changes between the clusters, Precision Interfaces would have created a single interface to express them all.

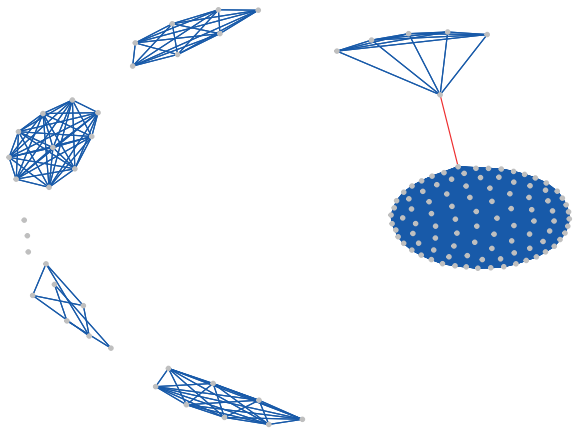

Figure 8: Transformation graph for one of the analysis sessions. Blue edges describe changes in the `SELECT` clause, red edges describe changes in the `WHERE` clause.

6 CONCLUSIONS AND FUTURE WORK

We have argued for *Precision Interfaces* and described our prototype system that generates such interfaces from program logs. We described a domain specific language for specifying interesting structural changes between program parse trees, modeled the program log as an *interaction graph*, and described a graph-based algorithm for mapping the graph to a set of interface components. Our case study on query traces generated from several open-ended Tableau exploration sessions showed that different users (and even the same user) perform different types of analysis tasks, and Precision Interfaces generated simple, custom interfaces for each task. This research is still in the early stages, and we are actively working on the following extensions to the system.

Optimizations. In practice, interaction graphs are extremely dense because most transformations are transitive. Consider changing the table name in the `FROM` clasue. If Q_1 can transform into Q_2, and Q_2 can transform into Q_3, then Q_1 certainly transforms into Q_3. Similarly, many transformations are also reflexive. This forms dense, strongly connected clusters in the graph with $O(N^2)$ edges for N queries.

We are exploring blocking-based techniques [4] that can avoid all pair-wise comparisons within a dense cluster of programs, as well as sampling techniques that can guarantee that the sampled interaction graph will result in equivalent generated interfaces.

Rendering. This paper described generating interface components that the user can use to express program changes, however we have actively not considered how program *outputs* should be rendered in the interface. A simple approach is to provide default tabular visualizations or use existing visualization generation techniques [12, 13, 24], however we are also exploring ways to identify the rendering functions in the programs themselves and incorporate them into the interface.

Incremental Maintenance. We envision running Precision Interfaces as a system process that monitors and recommends new interfaces automatically. In such a setting, the program log is constantly evolving and it is desirable to generate new or enhanced interfaces without re-running the whole pipeline. Similarly, it is desirable to identify and discard obsolete interfaces. We are exploring incremental approaches to dynamically maintaining the interaction graph [14] as well as the set of generated interfaces.

Automatic PILang. The quality of the generated interfaces depends on a rich set of PILang statements that represent the core set of structural changesin the log. Identifying and specifying these statements is a key challenge. We are working on automatically inferring PILang statements from program logs and richer interface component specifications. For instance, consider a simple slider—it is parameterized by the minimum and maximum numbers, and can modify a single number. This specification naturally restricts the classes of PILang statements that it can map to. Similarly, we might not consider complex strutural changes such as adding and removing quantification expressions because the only interface components that may express those are text boxes or specially crafted interface components.

Acknowledgements: We thank Yifan Wu, who provided the initial inspiration for this project, and Laura Rettig who worked on early formulations of the problem.

REFERENCES

[1] S. Alspaugh, B. Di Chen, J. Lin, A. Ganapathi, M. A. Hearst, and R. H. Katz. Analyzing log analysis: An empirical study of user log mining. In *LISA*, 2014.
[2] S. Alspaugh, A. Ganapathi, M. A. Hearst, and R. Katz. Better logging to improve interactive data analysis tools. In *KDD Workshop on Interactive Data Exploration and Analytics (IDEA)*, 2014.
[3] M. J. Bates. The design of browsing and berrypicking techniques for the online search interface. *Online Information Review*, pages 407–424, 1989.
[4] R. Baxter, P. Christen, T. Churches, et al. A comparison of fast blocking methods for record linkage. In *KDD*, 2003.
[5] G. E. Evans and M. G. Simkin. What best predicts computer proficiency? *Communications of the ACM*, pages 1322–1327, 1989.
[6] K. Z. Gajos, D. S. Weld, and J. O. Wobbrock. Automatically generating personalized user interfaces with supple. *Artificial Intelligence*, 174(12-13):910–950, 2010.
[7] J. Heer, J. M. Hellerstein, and S. Kandel. Predictive interaction for data transformation. In *CIDR*, 2015.
[8] J. M. Hellerstein, V. Sreekanti, J. E. Gonzalez, J. Dalton, A. Dey, S. Nag, K. Ramachandran, S. Arora, A. Bhattacharyya, S. Das, M. Donsky, G. Fierro, C. She, C. Steinbach, V. Subramanian, and E. Sun. Ground: A data context service. In *CIDR*, 2017.
[9] S. Idreos, O. Papaemmanouil, and S. Chaudhuri. Overview of data exploration techniques. In *Proc. SIGMOD*, pages 277–281, 2015.
[10] W. C. Kim and J. D. Foley. Providing high-level control and expert assistance in the user interface presentation design. In *Proc. INTERACT'93 and CHI'93*, pages 430–437, 1993.
[11] H. Lam. A framework of interaction costs in information visualization. *IEEE TVCG*, 14(6), 2008.
[12] J. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Transactions On Graphics*, pages 110–141, 1986.
[13] J. Mackinlay, P. Hanrahan, and C. Stolte. Show me: Automatic presentation for visual analysis. *IEEE TVCG*, 2007.
[14] J. Mondal and A. Deshpande. Managing large dynamic graphs efficiently. In *SIGMOD*, 2012.
[15] B. Myers, S. E. Hudson, and R. Pausch. Past, present, and future of user interface software tools. *ACM TOCHI*, 7(1):3–28, 2000.
[16] D. A. Norman. *The psychology of everyday things*. Basic books, 1988.
[17] A. Parush, R. Nadir, and A. Shtub. Evaluating the layout of graphical user interface screens: Validation of a numerical computerized model. *International Journal of Human-Computer Interaction*, 1998.
[18] B. Schneiderman. Eight golden rules of interface design. *Disponible en*, 1986.
[19] B. Shneiderman. *Designing the user interface: strategies for effective human-computer interaction*. Pearson Education India, 2010.
[20] J. M. Spool. What makes a design seem 'intuitive'. *User Interface Engineering*, 2005.
[21] I. Square. Crossfilter: Fast multidimensional filtering for coordinated views, 2013.
[22] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):52–65, 2002.
[23] D. Weld, C. Anderson, P. Domingos, O. Etzioni, K. Z. Gajos, T. Lau, and S. Wolfman. Automatically personalizing user interfaces. In *Proc. IJCAI*. ACM, 2003.
[24] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer. Voyager: Exploratory analysis via faceted browsing of visualization recommendations. *IEEE TVCG*, pages 649–658, 2016.
[25] T. Yeh, T.-H. Chang, and R. C. Miller. Sikuli: using gui screenshots for search and automation. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, pages 183–192, 2009.
[26] B. V. Zanden and B. A. Myers. Automatic, look-and-feel independent dialog creation for graphical user interfaces. In *Proc. SIGCHI*, pages 27–34. ACM, 1990.