

Proceedings

2018 IEEE Symposium on
Visual Languages and Human-Centric Computing
(VL/HCC)

VL/HCC 2018

Edited By

Jácome Cunha
João Paulo Fernandes
Caitlin Kelleher
Gregor Engels
Jorge Mendes

October 1 – 4, 2018
Lisbon, Portugal

ISBN 978-1-5386-4235-1
IEEE Catalog Number CFP18060-ART

2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)

Copyright © 2018 by the Institute of Electrical and Electronics Engineers, Inc. All rights reserved.

Copyright and Reprint Permission: Abstracting is permitted with credit to the source. Libraries are permitted to photocopy beyond the limit of U.S. copyright law for private use of patrons those articles in this volume that carry a code at the bottom of the first page, provided the per-copy fee indicated in the code is paid through Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923. For reprint or republication permission, email to IEEE Copyrights Manager at pubs-permissions@ieee.org. All rights reserved. Copyright © 2018 by IEEE.

IEEE Catalog Number CFP18060-ART

ISBN 978-1-5386-4235-1

ISSN 1943-6106

Additional copies of this publication are available from

Curran Associates, Inc.
57 Morehouse Lane
Red Hook, NY 12571 USA
+1 845 758 0400
+1 845 758 2633 (FAX)
email: curran@proceedings.com

Table of Contents

Foreword	vii
Organizing Committee	x
Sponsors	xi
Supporters	xii

Keynotes

<i>Helping developers with privacy</i>	1
Jason Hong	
<i>Mind the gap: Modelling the human in human-centric computing</i>	3
Geraldine Fitzpatrick	
<i>Bringing visual languages to market: The OutSystems story</i>	5
Rodrigo Sousa Coutinho	

Socio-Technical Tools and Analyses

<i>Comparative Visualizations through Parameterization and Variability</i>	7
Karl Smeltzer and Martin Erwig	
<i>Creating Socio-Technical Patches for Information Foraging: A Requirements Traceability Case Study</i>	17
Darius Cepulis and Nan Niu	
<i>Semi-Automating (or not) a Socio-Technical Method for Socio-Technical Systems</i>	23
Christopher Mendez, Zoe Steine Hanson, Alannah Oleson, Amber Horvath, Charles Hill, Claudia Hilderbrand, Anita Sarma and Margaret Burnett	
<i>Searching Over Search Trees for Human-AI Collaboration in Exploratory Problem Solving: A Case Study in Algebra</i>	33
Benjamin T. Jones and Steven L. Tanimoto	

Improving Programmer Efficiency

<i>Expresso: Building Responsive Interfaces with Keyframes</i>	39
Rebecca Krosnick, Sang Won Lee, Walter S. Lasecki and Steve Oney	
<i>The design and evaluation of a gestural keyboard for entering programming code on mobile devices</i>	49
Gennaro Costagliola, Vittorio Fuccella, Amedeo Leo, Luigi Lomasto and Simone Romano	

<i>Evaluation of A Visual Programming Keyboard on Touchscreen Devices</i>	57
Islam Almusaly, Ronald Metoyer and Carlos Jensen	
<i>CodeDeviant: Helping Programmers Detect Edits That Accidentally Alter Program Behavior</i>	65
Austin Z. Henley and Scott D. Fleming	
Supporting End User Programmers	
<i>End-User Development in Social Psychology Research: Factors for Adoption</i>	75
Daniel Rough and Aaron Quigley	
<i>Calculation View: multiple-representation editing in spreadsheets</i>	85
Advait Sarkar, Andrew D. Gordon, Simon Peyton Jones and Neil Toronto	
<i>No half-measures: A study of manual and tool-assisted end-user programming tasks in Excel</i>	95
Rahul Pandita, Chris Parnin, Felienne Hermans, Emerson Murphy-Hill	
<i>APPINITE: A Multi-Modal Interface for Specifying Data Descriptions in Programming by Demonstration Using Natural Language Instructions</i>	105
Toby Jia-Jun Li, Igor Labutov, Xiaohan Nancy Li, Xiaoyi Zhang, Wenze Shi, Wanling Ding, Tom M. Mitchell and Brad A. Myers	
Understanding and Supporting Learning	
<i>The Impact of Culture on Learner Behavior in Visual Debuggers</i>	115
Kyle Thayer, Philip J. Guo and Katharina Reinecke	
<i>Tinkering in the Wild: What Leads to Success for Female End-User Programmers?</i>	125
Louise Ann Lyon, Chelsea Clayton and Emily Green	
<i>Exploring the Relationship Between Programming Difficulty and Web Accesses</i>	131
Duri Long, Kun Wang, Jason Carter and Prasun Dewan	
<i>A Large-Scale Empirical Study on Android Runtime-Permission Rationale Messages</i>	137
Xueqing Liu, Yue Leng, Wei Yang, Wenyu Wang, Chengxiang Zhai and Tao Xie	
Next Generation Tools	
<i>Interactions for Untangling Messy History in a Computational Notebook</i>	147
Mary Beth Kery and Brad A. Myers	
<i>Supporting Remote Real-Time Expert Help: Opportunities and Challenges for Novice 3D Modelers</i>	157
Parmit K. Chilana, Nathaniel Hudson, Srinjita Bhaduri, Prashant Shashikumar and Shaun Kane	
<i>ZenStates: Easy-to-Understand Yet Expressive Specifications for Creative Interactive Environments</i>	167
Jeronimo Barbosa, Marcelo M. Wanderley and Stéphane Huot	
<i>It's Like Python But: Towards Supporting Transfer of Programming Language Knowledge</i>	177
Nischal Shrestha, Titus Barik and Chris Parnin	

Modeling

<i>Automatic Layout and Label Management for Compact UML Sequence Diagrams</i>	187
Christoph Daniel Schulze, Gregor Hoops and Reinhard von Hanxleden	
<i>Evaluating the efficiency of using a search-based automated model merge technique</i>	193
Ankica Barišić, Csaba Debreceni, Daniel Varro, Vasco Amaral and Miguel Goulão	
<i>SiMoNa: A Proof-of-concept Domain-Specific Modeling Language for IoT Infographics</i>	199
Cleber Matos de Moraes, Judith Kelner, Djamel Sadok and Theo Lynn	
<i>Visual Modeling of Cyber Deception</i>	205
Cristiano De Faveri and Ana Moreira	

Supporting Data Science

<i>Milo: A visual programming environment for Data Science Education</i>	211
Arjun Rao, Ayush Bihani and Mydhili Nair	
<i>A Usability Analysis of Blocks-based Programming Editors using Cognitive Dimensions</i>	217
Robert Holwerda and Felienne Hermans	
<i>Stream Analytics in IoT Mashup Tools</i>	227
Tanmaya Mahapatra, Christian Prehofer, Ilias Gerostathopoulos and Ioannis Varsamidakis	
<i>BONNIE: Building Online Narratives from Noteworthy Interaction Events</i>	233
Vinícius Segura, Juliana Jansen Ferreira and Simone D. J. Barbosa	

APIs and Use of Programming Languages

<i>What Programming Languages Do Developers Use? A Theory of Static vs Dynamic Language Choice</i>	239
Aaron Pang, Craig Anslow and James Noble	
<i>API Designers in the Field: Design Practices and Challenges for Creating Usable APIs</i>	249
Lauren Murphy, Mary Beth Kery, Oluwatosin Alliyu, Andrew Macvean and Brad A. Myers	
<i>DeployGround: A Framework for Streamlined Programming from API Playgrounds to Application Deployment</i>	259
Jun Kato and Masataka Goto	

Graduate Consortium

<i>Human-AI Interaction in Symbolic Problem Solving</i>	265
Benjamin T. Jones	
<i>Supporting Effective Strategies for Resolving Vulnerabilities Reported by Static Analysis Tools</i>	267
Justin Smith	
<i>The novice programmer needs a plan</i>	269
Kathryn Cunningham	
<i>Using Program Analysis to Improve API Learnability</i>	271
Kyle Thayer	

<i>Towards Scaffolding Complex Exploratory Data Science Programming Practices</i>	273
Mary Beth Kery	
<i>Towards Supporting Knowledge Transfer of Programming Languages</i>	275
Nischal Shrestha	
<i>Creating Interactive User Interfaces by Demonstration using Crowdsourcing</i>	277
Rebecca Krosnick	
<i>Assisting the Development of Secure Mobile Apps with Natural Language Processing</i>	279
Xueqing Liu	
<i>Using Electroencephalography (EEG) to Understand and Compare Students' Mental Effort as they Learn to Program Using Block-Based and Hybrid Programming Environments</i>	281
Yerika Jimenez	

Showpieces

<i>The GenderMag Recorder's Assistant</i>	283
Christopher Mendez, Andrew Anderson, Brijesh Bhuvia and Margaret Burnett	
<i>Fritz: A Tool for Spreadsheet Quality Assurance</i>	285
Patrick Koch and Konstantin Schekotihin	
<i>Code review tool for Visual Programming Languages</i>	287
Giuliano Ragusa and Henrique Henriques	
<i>Automated Test Generation Based on a Visual Language Applicational Model</i>	289
Mariana Cabeda and Pedro Santos	
<i>HTML Document Error Detector and Visualiser for Novice Programmers</i>	291
Steven Schmoll, Anith Vishwanath Meeduturi, Mohammad Ammar Siddiqui, Boppaiah Koothanda Subbaiah and Caslon Chua	
<i>Toward an Efficient User Interface for Block-Based Visual Programming</i>	293
Yota Inayama and Hiroshi Hosobe	

Posters

<i>Human-Centric Programming in the Large - Command Languages to Scalable Cyber Training</i>	295
Prasun Dewan, Blake Joyce and Nirav Merchant	
<i>Visual Knowledge Negotiation</i>	299
Alan Blackwell, Luke Church, Matthew Mahmoudi and Mariana Mărăsoiu	
<i>A Modelling Language for Defining Cloud Simulation Scenarios in RECAP Project Context</i>	301
Cleber Matos de Moraes, Patricia Endo, Sergej Svorobej and Theo Lynn	
<i>A Vision for Interactive Suggested Examples for Novice Programmers</i>	303
Michelle Ichinco	
<i>An Exploratory Study of Web Foraging to Understand and Support Programming Decisions</i>	305
Jane Hsieh, Michael Xieyang Liu, Brad A. Myers and Aniket Kittur	

<i>Graphical Visualization of Difficulties Predicted from Interaction Logs</i>	307
Duri Long, Kun Wang, Jason Carter and Prasun Dewan	
<i>How End Users Express Conditionals in Programming by Demonstration for Mobile Apps</i>	311
Marissa Radensky, Toby Jia-Jun Li and Brad A. Myers	
<i>Educational Impact of Syntax Directed Translation Visualization, a Preliminary Study</i>	313
Damián Nicolalde-Rodríguez and Jaime Urquiza-Fuentes	
<i>Semantic Clone Detection: Can Source Code Comments Help?</i>	315
Akash Ghosh and Sandeep Kaur Kuttal	
<i>What Makes a Good Developer? An Empirical Study of Developers' Technical and Social Competencies</i>	319
Cheng Zhou, Sandeep Kaur Kuttal and Iftekhar Ahmed	
<i>Visualizing Path Exploration to Assist Problem Diagnosis for Structural Test Generation</i>	323
Jiayi Cao, Angello Astorga, Siwakorn Srisakaokul, Zhengkai Wu, Xueqing Liu, Xusheng Xiao and Tao Xie	
<i>Usability Challenges that Novice Programmers Experience when Using Scratch for the First Time</i>	327
Yerika Jimenez, Amanpreet Kapoor and Christina Gardner-McCune	
<i>BioWebEngine: A generation environment for bioinformatics research</i>	329
Paolo Bottoni, Tiziana Castrignanò, Tiziano Flati and Francesco Maggi	

Foreword

VL/HCC 2018

We were pleased to welcome delegates to the 2018 IEEE Symposium on Visual Languages and Human Centric Computing, held in Lisbon, Portugal at the Universidade NOVA de Lisboa. The theme of 2018's conference was Building Human-Adaptive Socio-Technical Systems. These kinds of systems incorporate humans as both developers of and intrinsic parts of the system. We invited three keynote speakers, two from academia and one from industry following this theme. The first, Jason Hong, is a Professor in the Human-Computer Interaction Institute at Carnegie Mellon University. His research focuses on human-computer interaction and privacy and security, a combination that is exposed in new ways in Socio-Technical Systems. His talk focused on understanding and building new tools that address developers' issues with supporting privacy. The second keynote was given by Geraldine Fitzpatrick who is a Professor of Technology Design and Assessment and leads the Human Computer Interaction Group at TU Wien in Vienna, Austria. Her research explores the intersection of social and computer sciences. Her talk uses the domain of developing supportive technologies for aging people to expose issues around modelling behaviors for systems and the realities of living with those models. The final keynote, from industry, was given by Rodrigo Sousa Coutinho, the co-founder and Strategic Product Manager at OutSystems, a Portugal-based software firm which constructed a platform that transforms visual models into running enterprise-grade applications. His talk focused on the story of bringing their visual language to market, and highlighted how OutSystems collaborated with academia.

The VL/HCC 2018 included calls for research papers, full and short (also known as work in progress), posters and showpieces (including tools and other artifacts demonstrations). We also included a call for workshops and tutorials. Research papers were allowed to submit any additional material authors would see as interesting to complement the paper submission, including a video. Several authors submitted such material, which will also be available with the proceedings.

For research papers, the 2018 edition of VL/HCC kept the lightweight double blind review process. The submissions were anonymized as well as the reviews. However, during the rebuttal phase which was also included this year, the authors' anonymity was removed, thus allowing to resolve eventual authorship issues. Each paper was reviewed by at least 3 PC members. After the rebuttal phase, there was a virtual PC meeting where each paper was discussed. To be noted that all PC members were allowed to participate in such a discussion, thus making it richer and broader.

As a best practice the General and Program Committee chairs did not submit any kind of contribution. Other chairs also did not submit contributions to their own tracks.

VL/HCC 2018 attracted 66 research paper submissions, 51 of which within the long paper category and the remaining 15 within the short paper category. Of these, the program committee decided to accept 19 long papers, yielding a 29% acceptance rate. In addition, 5 short papers were accepted, and 7 long papers were converted to short papers, so the program ended up including 12 short papers.

The technical program was also complemented with showpieces, a poster session and a graduate consortium.

The showpieces track accepted 6 contributions, whereas the poster track accepted 8. In addi-

tion, 5 research paper submissions were converted to posters, so the program included 13 posters. The graduate consortium accepted 9 student contributions.

We received 2 workshop submissions, both accepted, namely the *5th International Workshop on Software Engineering Methods in Spreadsheets* and the *1st International Workshop on Designing Technologies to Support Human Problem Solving*. Unfortunately, we did not receive any tutorial proposal.

We believe that the series of events and presentations that were put together provided for an excellent environment for the community to engage and improve our fields of research, which are broadly represented in a remarkable way. Also, the venue definitely provided exposure for works that are in significantly heterogeneous phases, ranging from early stage research and doctorate proposals to fully mature research contributions with potential to influence the state of the practice.

We are much obliged to the individual contributions that shaped what we expect to be generally recognized as a wonderful event. We are particularly grateful to the members of the Program Committee whose commendable dedication resulted in extensive and insightful reviews and enriching follow up discussions. Complementary, we would like to acknowledge the 232 authors of all the different types of submissions, for their leading research and their willingness to write papers describing it.

Several people were directly involved in the organization of VL/HCC 2018: João Saraiva and Stefan Sauer, as Workshop Co-chairs; Birgit Hofer and Miguel Goulão as Showpieces Co-chairs; Licínio Roque and Sandeep Kuttal as Poster Co-chairs; Scott Fleming and Vasco Amaral as Graduate Consortium Co-chairs; Jorge Mendes as Proceedings Chair; Marco Couto and Rui Pereira as Web and Registration Co-chairs; Ana Moreira and João Araújo as Finance and Local Organization Co-chairs; João Miguel Fernandes as Sponsor Chair; and Justin Smith as Social Media Chair. We would publicly like to express our gratitude for your excellent service and availability.

We would also like to acknowledge the significance of the support we received from our sponsors, namely OutSystems, Luso-American Development Foundation, Turismo de Lisboa and IEEE Portugal Section, and from our supporters, namely Universidade de Coimbra & CISUC, Universidade do Minho, Universidade NOVA de Lisboa, FCT & NOVA LINCS and IEEE.

We have worked hard for the conference to provide for an exciting and stimulating environment to discuss and further progress our scientific areas, and to consolidate existing collaborations as well as to potentiate fresh ones. It is with renovated confidence that we hope to have achieved such goal. Moreover, we sincerely hope that Lisboa, the Portuguese hospitality, its culture and gastronomy, and the joyful way of life of the Portuguese people provided you all a memorable and treasurable experience!

Jácome Cunha and João Paulo Fernandes
General Co-chairs

Caitlin Kelleher and Gregor Engels
Program Co-chairs

Organizing Committee

General Chairs

Jácome Cunha, Universidade do Minho
João Paulo Fernandes, Universidade de Coimbra

Program Committee Chairs

Caitlin Kelleher, Washington University
Gregor Engels, Paderborn University

Workshop Chairs

João Saraiva, Universidade do Minho
Stefan Sauer, Paderborn University

Showpieces Chairs

Miguel Goulão, Universidade Nova de Lisboa
Birgit Hofer, Graz University of Technology

Poster Chairs

Sandeep Kuttal, University of Tulsa
Licínio Gomes Roque, Universidade de Coimbra

Graduate Consortium Chairs

Scott Fleming, University of Memphis
Vasco Amaral, Universidade Nova de Lisboa

Proceedings Chair

Jorge Mendes, Universidade do Minho

Web & Registration Chairs

Marco Couto, Universidade do Minho
Rui Pereira, Universidade do Minho

Finance & Local Organization Chairs

Ana Moreira, Universidade Nova de Lisboa
João Araújo, Universidade Nova de Lisboa

Sponsor Chair

João Miguel Fernandes, Universidade do Minho

Social Media Chair

Justin Smith, North Carolina State University

Sponsors

Silver



Bronze



Supporters



UNIVERSIDADE DE COIMBRA

CISUC



Universidade do Minho
Escola de Engenharia



NOVALINCS
LABORATORY FOR COMPUTER
SCIENCE AND INFORMATICS



Helping developers with privacy

(Invited Keynote)

Jason Hong
*Human-Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, PA, USA
jasonh@cs.cmu.edu*

ABSTRACT

The widespread adoption of smartphones and social media make it possible to collect sensitive data about people at a scale and fidelity never before possible. While this data can be used to offer richer user experiences, this same data also poses new kinds of privacy challenges for organizations and developers. However, developers often have little or no knowledge about how to design and implement for privacy.

In this talk, I discuss our team's research on helping developers with privacy. I will present some results of interviews and surveys with developers, as well as different tools we have developed. A key theme guiding our work is looking for ways of making developers' lives easier, while making privacy a positive side effect.

ABOUT THE SPEAKER

Jason Hong is an associate professor in the Human Computer Interaction Institute, part of the School of Computer Science at Carnegie Mellon University. He works in the areas of usability, mobility, privacy, and security.

He is an author of the book *The Design of Sites*, a popular book on web design using web design patterns. Jason was also a co-founder of Wombat Security Technologies, which focused on cybersecurity training, and was acquired by Proofpoint in 2018.

Jason received his PhD from Berkeley and his undergraduate degrees from Georgia Institute of Technology. Jason has participated on DARPA's Computer Science Study Panel (CS2P), is an Alfred P. Sloan Research Fellow, a Kavli Fellow, a PopTech Science fellow, a New America National Cybersecurity Fellow.

Mind the gap: Modelling the human in human-centric computing

(Invited Keynote)

Geraldine Fitzpatrick

TU Wien (Vienna University of Technology)

Vienna, Austria

geraldine.fitzpatrick@tuwien.ac.at

ABSTRACT

The topic of Human-Adaptive Socio-Technical Systems – requiring human-centered concepts, languages and methods to specify system behavior and to model human behavior – is increasingly important as these systems become complexly entangled in everyday lives and contexts.

However, the extent to which we achieve ‘good’ systems is very much shaped by what gets specified and modelled about the human in the system. Using the case of older people and care, I will draw attention to the gaps between modelling of behaviours for building systems and negotiating the lived realities of those behaviours, pointing to tensions around whose voice(s) count, what conceptualisations of ‘old’ and ‘care’ dominate, how competing values are negotiated or not, and who defines what adaptations are appropriate. At an application level I want to argue for enabling older people to become active co-contributors to care networks rather than passive recipients of care, and designing for living not aging.

More generally though I want to argue for the critical importance of growing our own social and emotional competencies to more sensitively engage with the complexities Human-Adaptive Socio-Technical Systems and embrace an explicit concern for the values we are modelling into our systems.

ABOUT THE SPEAKER

Geraldine Fitzpatrick is Professor of Technology Design and Assessment and heads the Human Computer Interaction Group at TU Wien in Vienna, Austria. Previously, she was Director of the Interact Lab at Uni. of Sussex, User Experience consultant at Sapient London, and Senior Researcher at the Distributed Systems Technology CRC in Australia.

Her research is at the intersection of social and computer sciences, with a particular interest in technologies supporting collaboration, health and well-being, social and emotional skills learning, and community building. She has a published book and over 180 refereed journal and conference publications in diverse areas such as HCI, CSCW, health informatics, pervasive computing.

She sits on various international faculty and project advisory boards, and serves in various editorial roles, and in program committee/chair roles at various CSCW/CHI/health related international conferences and is the Austrian representative at IFIP TC-13. She is also an ACM Distinguished Scientist and ACM Distinguished Speaker, and hosts the Changing Academic Life podcast series.

Bringing visual languages to market: The OutSystems story

(Invited Keynote)

Rodrigo Sousa Coutinho
OutSystems
Portugal

ABSTRACT

In 2001, OutSystems was created with the goal of helping enterprises deliver applications on time and on budget. In order to achieve this ambitious goal, we built a platform from scratch that transforms visual models into running enterprise grade applications.

During this session, I will share how the market has grown around low-code platforms supported by visual languages. I will also tell the story behind the OutSystems visual language, and how we collaborated with academia to evolve the language to face the challenges and tradeoffs of delivering unique productivity gains to our developers - without compromising performance, security and robustness.

Finally, we'll look into the future and the challenges new types of users, like citizen developers, bring to a language. We'll also take a peek at how we can teach and guide users of visual languages to build high quality applications with the help of machine learning.

ABOUT THE SPEAKER

Rodrigo Coutinho is co-founder and Strategic Product Manager at OutSystems, and is currently responsible for strategizing the entire developer journey, from the first contact with OutSystems all the way to building complex enterprise grade applications.

Rodrigo has a strong participation in the design of the product, in particular its architecture and visual language, focusing on innovative and pragmatic ways to help increase the speed of delivery of enterprise applications.

Other roles at OutSystems included Head of Product Design and Principal Software Engineer. Before co-founding OutSystems in 2001, he was a Software Engineer at Altitude Software and Intervento.

Comparative Visualizations through Parameterization and Variability

Karl Smeltzer

Oregon State University

karl@lifetime.oregonstate.edu

Martin Erwig

Oregon State University

erwig@oregonstate.edu

Abstract—Comparative visualizations and the comparison tasks they support constitute a crucial part of visual data analysis on complex data sets. Existing approaches are ad hoc and often require significant effort to produce comparative visualizations, which is impractical especially in cases where visualizations have to be amended in response to changes in the underlying data.

We show that the combination of parameterized visualizations and variations yields an effective model for comparative visualizations. Our approach supports data exploration and automatic visualization updates when the underlying data changes. We provide a prototype implementation and demonstrate that our approach covers most of existing comparative visualizations.

I. INTRODUCTION

To make sense of the fast-growing amounts of data, information visualization is getting more and more important. The rate of data collection in general is growing exponentially, driven by the rise of technologies such as autonomous vehicles and smart devices [1]. In turn, this continues to drive the development of new approaches and techniques in data visualization to explore and explain the data.

Comparative visualization is one such approach that focuses specifically on comparison tasks when analyzing data. Comparisons tasks feature prominently in all kinds of visualization history mechanisms [2], uncertainty visualization [3], software visualization [4], and many more areas.

The widely adopted approach of using small multiples [5] (roughly, composing the variant visualizations into a grid containing all possibilities) provides only a partial solution. Two immediate problems of this approach are how to organize the charts into a grid and how to ensure they are simple enough to be read at a small size. More seriously, a grid layout is inherently only scalable in two dimensions. As the number of orthogonal parameters in the data grows we need exponentially many charts to keep up.

Another complication arises from the difficulty of context-dependent comparison. For example, suppose we are tasked with an analysis of profits per quarter for a business. During that process we might need to undertake some semantic zooming subtasks such as comparing only fourth quarter profits across years to see the impact of holiday bonuses, or perhaps exploring the data only for specific geographic regions. Such tasks can of course always be performed manually by returning to the data, subsetting or manipulating it, and then re-creating appropriate

visualizations. However, it is highly inefficient to essentially having to start from scratch with each iteration.

In this work we propose a model for creating, transforming, and comparing visualizations based on the notion of variation that helps to systematize how data scientists can approach these challenges. We begin by revisiting a model for constructing traditional, non-variational visualizations in Section II. In Section III we review a model of variation as well as its application to represent variational pictures. Built on these two components, we introduce variational visualizations in Section IV with numerous examples. In Section V we evaluate how our model of variational visualizations supports comparative visualization. Section VI discusses related work and Section VII presents some conclusions. This work makes the following specific contributions.

- An *expressive model* of variational visualizations.
- A *prototype* implementation of the model, used to generate all of the example figures in this paper.
- An *evaluation* of the model’s suitability for comparative visualization tasks.

II. A MODEL FOR DATA VISUALIZATION

We build on the visualization approach presented in [6]. Here we focus on a small subset of visualization types with the goal of systematizing how we construct and compose them.

A visualization is essentially a composition of *marks*. Marks encode primitive shapes implicitly through visual parameter mappings. Based on Bertin’s visual variable [7], visual parameters are any rendered aspects of a mark that can be bound to a data value, such as color, size, location, orientation, etc. Marks also contain labeling information.

$$\text{Mark} = \text{VisualParameter}^* \times \text{Label}$$

A visualization is essentially given by a composition of marks, a transformation between coordinate systems or an overlay of visualizations. For simplicity we employ a simple prefix notation (allowing binary operations to be written infix).

$$\begin{aligned} \text{Visualization} &::= \text{Mark} \\ &\mid \text{NextTo Visualization}^* \\ &\mid \text{Above Visualization}^* \\ &\mid \text{Cartesian Visualization} \\ &\mid \text{Polar Visualization} \\ &\mid \text{Overlay Visualization}^* \\ &\mid \dots \end{aligned}$$

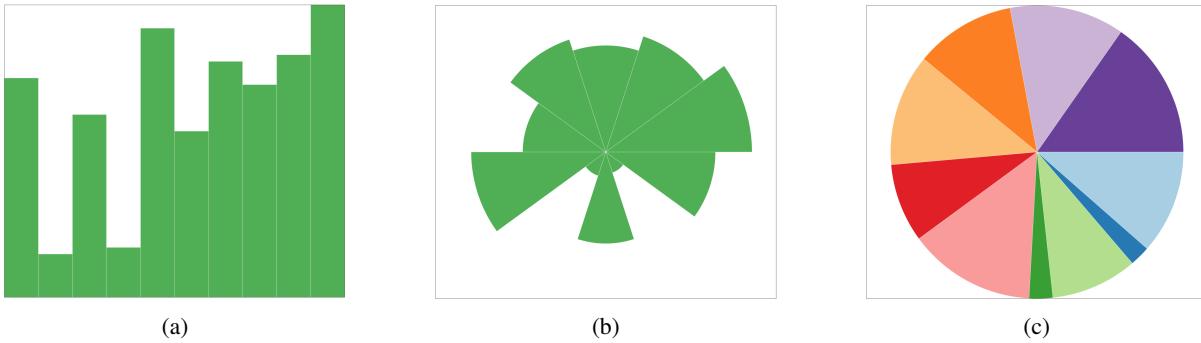


Fig. 1: A series of non-variational and non-comparative visualizations constructed in our prototype using composition and transformation. Both (b) and (c) are built from the visualization in (a) as a starting point.

In a Cartesian coordinate system the **NextTo** and **Above** constructs divide space horizontally and vertically, respectively. In a polar coordinate system they divide the space by angle and radius, respectively.

For example, to construct a bar chart we need a sequence of marks in which the height visual parameter of the primitive rectangles is bound to a data value. We can also set the color parameter of the marks and attach labels.

```
 $m_1 = ([height \mapsto 6.6, color \mapsto green, \dots], "6.6")$ 
 $\vdots$ 
 $m_{10} = ([height \mapsto 8.1, color \mapsto green, \dots], "8.1")$ 
```

We can then compose these horizontally to generate a simple chart shown in Figure 1a.¹

```
 $bars = \text{NextTo } [m_1, \dots, m_n]$ 
```

Our model provides many shortcuts to avoid the tedious construction of the individual marks. For example:

```
 $bars = \text{barchart } [6.6, \dots, 8.1] \text{ 'colorAll' green}$ 
```

Building visualizations from composable parts makes it easy to transform them. For example, suppose we would like to see whether a Coxcomb chart (essentially a radial area chart, where data is bound to the radius of equal-angle wedges) might be more appropriate than a bar chart. In a typical visualization tool this would require either starting over or perhaps copying and modifying the code responsible for generating it.

Instead, our model allows visualizations to be transformed directly, which avoids the need for managing multiple artifacts or code snippets. In this particular case, a Coxcomb chart is simply a bar chart reinterpreted in a polar coordinate system. The marks are still composed next to one another (because in a polar environment, horizontal composition corresponds to the angle), and the data bound to the height does not need to change (since vertical height corresponds to the radius distance in the polar system). The rendered output of this transformation is shown in Figure 1b.

```
 $cox = \text{Polar bars}$ 
```

¹All visualizations in this paper have been created with a prototype implementation of a visualization DSL that can be found at: <https://github.com/karljs/vis>.

If we find the Coxcomb chart too difficult to read, we can turn it into a pie chart instead. In a typical visualization tool we again would likely need to start over. But because a pie chart corresponds closely to a Coxcomb chart, we can produce one easily using another transformations. We still want to compose our marks next to one another in a polar environment, but we want to change the data to map to the angle (or width) instead of the radius (or height). Since this requires modifying the visual parameters, we need more than just another composition operator. For this purpose we provide a series of functions that modify visualizations, such as *reorient*, which flips the width and height parameters of all the marks in a visualization for us.

```
 $\text{reorient} : \text{Visualization} \rightarrow \text{Visualization}$ 
```

Applying the *reorient* operation gives us the pie chart we were expecting, but there is one more thing we can do. Pie charts are often more effective when the individual wedges are colored distinctly rather than with a single color, to provide some visual separation. We could choose new colors manually, but our system also defines some color schemes that can be applied automatically. We can use the *color* operation to recolor the chart, which takes a sequence of colors as a parameter. The simplest way to use a nice default qualitative color scheme we can just pass it the built-in *defaultColors*.

```
 $pie = \text{reorient cox 'color' defaultColors}$ 
```

This produces the rendered result shown in Figure 1c. There are many more useful transformations that are possible; more detail is provided in [6].

One important feature of the presented visualization model is its ability to define and apply functions, which allows visualization to be parameterized. Parameterization provides a dynamic form of variationalization.

A. Comparative Visualizations Without Variability

It is not always necessary to use variation to compare two visualization designs. Using the operation of our visualization model, we can also already generate a limited form of comparative visualizations using operators such as **NextTo** and **Above**. For example, we could compare a barchart *bar*

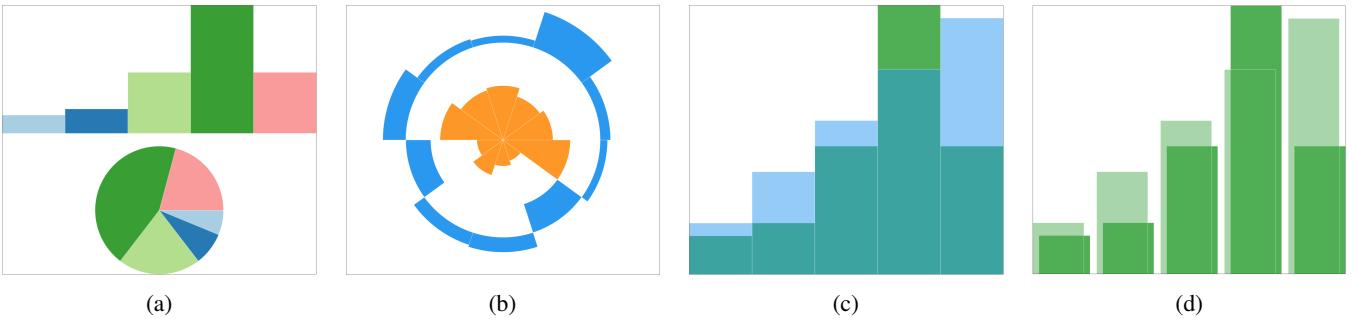


Fig. 2: Examples of comparative visualizations constructed in our prototype which do not make use of variability.

with its equivalent pie chart by explicitly placing one above the other, as shown in Figure 2a.

Above [bar, reorient (Polar bar)]

Finally, when using polar charts, such as Coxcomb charts, we can align wedges in concentric rings, when appropriate. This makes comparing two polar charts more reasonable than putting them beside one another.

Polar (Above) [cox₁, cox₂]

Such a chart can be seen in Figure 2b.

Instead of showing visualizations side-by-side, another possibility for comparison tasks is to show them overlaid. This can be achieved in the same way as spatial composition by using **Overlay**. However, to avoid the occlusion of parts of visualizations, we can employ transparency to ensure lower layers are always visible. An example is shown in Figure 2c.

Another approach is to partially offset the marks comprising the lower layer visualizations in order to prevent them from being totally obscured. In the following example, we make use of both approaches simultaneously. The *alpha* command configured the level of transparency for a visualization. Setting RGBA colors directly is also supported. The spacing is slightly more complicated. The relative spacing model is detailed in [6], but the important aspect to know here is that when space is applied it is always sized relative to a particular visualization or element. For example, if spacing is applied between the bars of a bar chart, it is specified as a ratio to the width of the bars. If spacing is inserted between entire visualizations, it is sized as a ratio of the width of the entire visualizations. In the following example, we apply spacing both between bars and on the edges of entire visualizations in order to provide enough space for all bars to show.

The *space* function places empty space between the elements of the visualization to which it is applied. In this case, we apply it to a composition of bars with the argument 0.25, which means that it will create a space equivalent to one-quarter the width of the bars between each pair of bars. The *rightSpace* and *leftSpace* functions behave slightly differently. Those essentially compose empty space onto one side of an entire visualization. For example, when we apply *rightSpace* with the argument 0.02, it produces space equal to 2 percent of the visualization's

width and composes it on the right. The internal spacing is unaffected.²

Overlay [bar₁ ‘alpha’ 0.5 ‘space’ 0.25 ‘rightSpace’ 0.02, bar₁ ‘space’ 0.25 ‘leftSpace’ 0.02]

Since our visualization model allows the definition of functions, we can identify patterns such as this and capture them in function definitions. Applying this visualization pattern to all bars leads to the result shown in Figure 2d.

III. REPRESENTING VARIATION

The choice calculus [8] is a formal model of variation built on the core concept of *choices*. Choices attach names, or *dimensions*, to a list of *alternatives*. For example, we can write a choice in dimension *A* between two variant numbers as *A*(1, 2). Choices can also be nested, as in *A*(*B*(1, 2), 3). In this paper we limit ourselves to binary choices for simplicity, since it is always possible to represent choices with more alternatives using a sequence of nested choices.

Each binary dimension *D* also leads to two *selectors*, *D.l* and *D.r*. Selectors indicate particular branches in that dimensions and can be used for *selection*, which reduces or eliminates variability. Selection is defined as follows where *s* ranges over selectors, *vx* and *vy* range over variational values, and *x* ranges over plain values.

$$\lfloor D\langle vx, vy \rangle \rfloor_s = \begin{cases} \lfloor vx \rfloor_s & \text{if } s = D.l \\ \lfloor vy \rfloor_s & \text{if } s = D.r \\ D\langle \lfloor vx \rfloor_s, \lfloor vy \rfloor_s \rangle & \text{otherwise} \end{cases}$$

$$\lfloor x \rfloor_s = x$$

For example, $\lfloor A\langle B(1, 2), 3 \rangle \rfloor_{B.r} = A\langle 2, 3 \rangle$. When multiple dimensions share choices they are *synchronized*, meaning that performing a selection for one automatically performs the same selection for all choices in that dimension.

Sets of selectors are called *decisions* and can be used to eliminate variation with repeated selection. For a decision $\delta = \{s_1, s_2, \dots, s_n\}$, we have $\lfloor vx \rfloor_\delta = \lfloor \lfloor \dots \lfloor vx \rfloor_{s_1} \dots \rfloor_{s_{n-1}} \rfloor_{s_n}$. Note that the order of selection is irrelevant. We employ the variational type constructor *V* to distinguish variational values from non-variational ones. For example, we write $3 : Int$ for plain integers and $A\langle 1, 2 \rangle : V(Int)$ for variational ones.

²The backquotes allow the writing of binary function as infix operators.

A. Variational Pictures

One application of the choice calculus is for the representation of variational pictures [9]. Variational pictures are structures that encode arbitrarily many different pixel-based pictures. If a traditional picture is modeled as a function from pixel grid locations to T values (often colors), $\text{Pic}_T = \text{Loc} \rightarrow T$, then we can understand a variational picture to be a function from pixel grid locations to variational T values.

$$\text{VPic}_T = \text{Loc} \rightarrow V(T)$$

This allows us to define variational pictures by wrapping the pixels that vary in choices. For example, we could construct a small four-pixel variational picture $v = \circlearrowleft^A(B(\bullet, \star), \circlearrowright)$.

The two left pixels do not vary, while the two right pixels do. The top-right pixel varies in both the A and B dimensions, while the bottom-right pixel varies only in the A dimension. Because we have two dimensions we can be selected independently, this variational picture encodes four variant pictures (see below). The semantics of variational pictures is a mapping from decisions to plain pictures.

$$[\![\cdot]\!] : \text{VPic} \rightarrow V(\text{Pic})$$

$$[\![vp]\!] = \{(\delta, (l, x)) \mid (l, vx) \in vp, (\delta, x) \in vx\}$$

With this definition we can get the variants of the variational picture v as $\{(\{A.l, B.l\}, \circlearrowleft), (\{A.l, B.r\}, \circlearrowright), (\{A.r\}, \circlearrowleft), (\{A.r\}, \circlearrowright)\}$.

We employ the idea behind variational pictures as a model for representing visualizations that include variation. An important difference is that we do not represent variational pixels but apply variation to basic and composed visualization objects.

IV. VARIATIONAL VISUALIZATIONS

A variational visualization is a data visualization that encodes arbitrarily many different plain visualizations and provides a mechanism to navigate through all of the encoded variants. The differences among the encoded visualizations could be aesthetic such as colors or labels, they could be in terms of how the visual parameters are bound, in terms of the data being visualized, or some combination of these factors. To reason about all these possibilities we need to manage the variation in a systematic way.

A. Understanding Different Variability Designs

The visualization model described in Section II provides only limited support for comparison tasks. In particular, without an explicit representation for variation, the opportunities for navigating and manipulating variational visualizations is rather limited. On the other hand, the model of variational pictures in Section III-A is too limited to handle the transformation of variational visualizations. For that, we need yet another application of the core ideas in the choice calculus.

To illustrate this, consider using the variational type constructor V , introduced in Section III, to make arbitrary types variational. It might look something like this.

$$\begin{aligned} V(a) &::= D(V(a), V(a)) \\ &\quad | \quad a \end{aligned}$$

A value of type $V(a)$ is either a plain value, or a choice of two nested $V(a)$ values. A $V(a)$ value is a binary tree where the nodes are choices and the leaves are values of type a . This definition allows the top-level application of V to the visualization type defined in Section II, that is, a variational visualization would have the type $V(\text{Visualization})$. For example, We can construct the variational chart $\text{PICKCOLOR}\langle\text{blueChart}, \text{greenChart}\rangle$, which allows the selection between two charts as a whole, but it does not support comparing parts of two visualizations in context. This can be important if two visualizations are similar overall but differ only in a few places.

In addition to this top-level application of variability, there are two other possibilities to integrate V into structures [10], namely at the leaves or recursively.

Application of the V type constructor at the leaves involves moving the variation into the visualization structure, applying it directly to the marks. We could redefine our visualization type to be the following.

$$\begin{aligned} \text{Visualization} &::= V(\text{Mark}) \\ &\quad | \quad \text{NextTo Visualization}^* \\ &\quad | \quad \dots \end{aligned}$$

Since we likely want to avoid constructing all of our variational marks manually, we would need to update many of the built-in operations such as $barchart$. In order to construct a variational visualization where the marks are the parts that vary, we need to provide $barchart$ with variational data as input. That is, instead of the type

$$barchart : \text{List}(\text{Number}) \rightarrow \text{Visualization}$$

we would have something of the form

$$barchart : V(\text{List}(\text{Number})) \rightarrow \text{Visualization}$$

This new $barchart$ function would then be responsible for creating a mark for each variant data value.

By pushing the type constructor down into the marks we are able to eliminate two of the major drawbacks from the top-level approach: First, redundant visualization structures can be avoided because the variation can only occur at the innermost level, and second, we can also determine exactly where the variation occurs by observing the marks and their variation directly.

However, the leaf-level application of V prevents many kinds of useful variations in visualizations. For example, we are not able to represent a bar chart that is either a vertically or a horizontally oriented bar chart depending on the selection. These have subtly different structures (**Above** as opposed to **NextTo**), and since we only allow marks to vary, and not the composition operators, this is not possible. Therefore, this approach is obviously too limited to be useful in the general case.

A final possibility is to integrate the variability directly into the recursive structure of the visualizations, allowing it to occur wherever is most appropriate for the desired effect. We can add a new case to the visualization definition as follows.

$$\begin{aligned} \text{Visualization} &::= \dots \\ &\quad | \quad V(\text{Visualization}) \end{aligned}$$

The added flexibility of this recursive application of V allows us to avoid any issue with being unable to represent particular kinds of variation. Moreover, assuming the variation is allocated judiciously, we can also avoid unnecessary redundancy.

However, the burden of choosing where to integrate variation is shifted onto the visualization author. Given such a system, the user must have a sufficient understanding of its inner workings to not only know when it is preferable to move the variability inward or outward in the visualization structure, but also how to actually achieve this by using and defining operations. Still, given the drawbacks of the top-level and leaf-level approaches, the additional demands on the user seem to be reasonable.

B. Rendering and Navigating Variational Visualizations

Having found a suitable way to integrate variability into our visualizations, we still need to decide how they can be presented to and navigated by users. We have implemented a prototype, which renders variational visualizations according to the model of variational pictures described in Section III-A, that is, it produces a variational picture where each variant plain picture is one of the variant plain visualizations. All of the figures used in this paper have been generated by this prototype.

A tool for displaying variational visualizations must allow users to navigate among the different variants. For simplicity, we chose a simple approach based on standard GUI elements. We extract all of the dimensions that a variational visualization contains and produce a checkbox for each. This checkbox toggles the selection in that dimension. When one of the dimensions is toggled on, radio buttons are shown to select between either the left or right alternatives. This scheme allows users to specify any decision, whether partial or total, and see the rendered result.

When a decision is not total, and variation still exists in the visualization being rendered, it is not obvious what should be drawn. One possibility is to draw nothing for the parts that are unselected and just indicate that a selection must be made first, such as by a box or outline. Since we are focusing on comparison tasks, however, we chose an approach in which all variants of the current visualization are shown at once, side-by-side, using a small multiples approach. This not only supports comparison but also gives users the ability to see visually how much variation remains unselected in their visualization.

We have also used colors to map the navigation interface to the portions of the rendering canvas that they affect. For example, if a particular dimension toggles the height of a bar, we outline that bar's space with a colored, dashed outline and color the corresponding UI elements with the same color. These colors are assigned automatically. A screenshot of the prototype is shown in Figure 3.

C. Variational Comparative Visualizations

The simplest example of a variational visualization is to combine two existing visualizations in a choice, as indicated above with the choice between the green and blue barchart.

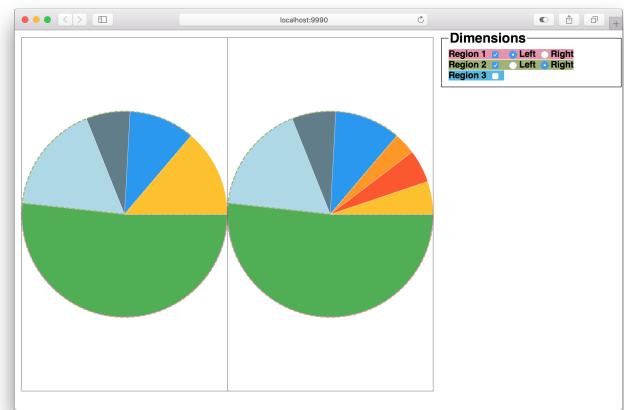


Fig. 3: A screen capture of our prototype user interface showing possible configuration/selection options. On the left is the rendered visualization currently being constructed and on the right are the interface elements which allow the user to navigate among the variants.

Recall from Section III that choice expressions can be simplified by selection. However, if the selection is performed with a decision that is not total, i.e., that does not map every choice to a particular alternative, then the variation is not entirely removed. In our prototype, when variation is not entirely eliminated with selection, we render all of the remaining variants in a small multiples grid.

Another use of variation is to control the level of detail shown for a set of data. For example, suppose we want to produce a pie chart showing a breakdown of some costs for geographic regions in the United States. We might want to show an overview for areas such as West coast, East coast, South, etc. (Figure 4a). However, we also want to make the details of the states comprising each region available on demand by selecting corresponding variants (Figure 4b). We can encode the zoomed out and zoomed in versions of each pie wedge in a choice, allowing variation to take care of the exponentially many different versions.³ The rendered output is shown in Figure 4.

```
Polar (NextTo [West<2.0, NextTo [1.0, 0.4, 0.6]>
                  East<0.8, NextTo [0.2, 0.3, 0.3]>
                  South<3.0, NextTo [1.9, 0.7, 0.4]>))
```

Variation is not just useful for comparing aesthetic options, however. One of the major advantages that an approach based on variation gives us is the ability to work directly with variational data. Suppose we want to examine source code that is annotated with C-preprocessor macros such as `#ifdef` to chart the number of lines of code in each block. We could produce each of the 2^n possible configurations (where n is the number of configuration options), count the relevant lines of code in each, and then produce a separate chart for each configuration.

³Some details regarding how the relative widths of nested visualization components are computed and the color assignment are omitted here for simplicity.

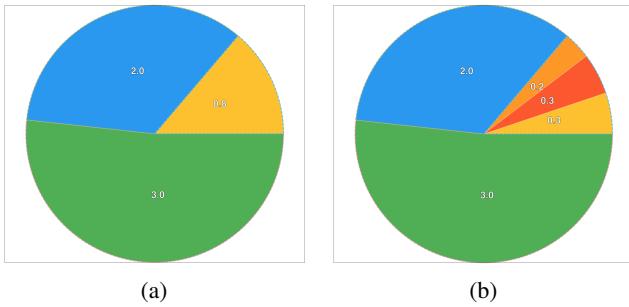


Fig. 4: Comparative visualizations for exploring visualization details; (a) Summary visualization that corresponds to the decision $\{\text{West.l}, \text{East.l}, \text{South.l}\}$; (b) Revealing details for the east with decision $\{\text{West.l}, \text{East.r}, \text{South.l}\}$.

This, however, is generally infeasible since large software projects have hundreds or even thousands of configuration options [11]. However, if we instead count the number of lines directly, but encode the values as variational numbers corresponding to the preprocessor macros, we can perform all of the work in a single pass. Since the data and the visualization tool would be making use of the same variation representation, we can then just chart the results directly. For example, we could make use of the *vbarchart* function, which is the variational equivalent of *barchart*.

vbarchart : $V(\text{Number}) \rightarrow \text{Visualization}$

The viewer of the visualization can then navigate the different configurations to compare the results for each.

Finally, we can compute modified versions of visualizations to compare between. Suppose, for example, we have two bar charts showing monthly earnings over the past two years for a business. We would now like to compare the two years directly by charting the change from the first year to the second for each month. We can use the *zipWith* function, which accepts two visualizations as input, traverses them in parallel, and computes a new visualization element based on a binary operation. In this example, we want to subtract the height of the bars in the second chart from the height of the bars in the first. We can simply use *zipWith* together with the $(-)$ function, which determines which visual parameter(s) are bound to data and computes their differences. For cases in which several visual parameters are bound to data, users can specify exactly which should be acted upon.

zipWith $(-)$ $\text{bar}_1 \text{ bar}_2$

An example of this computed visualization, composed next to the two original charts, is shown in Figure 5a.

Similarly, we can also compute data transformations directly on visualizations. For example, we may have a set of data already charted for which we also want to see both the log transformed version and a square root transformed version. Moreover, we want to see the original and transformed data at the same time, overlaid using transparency. We can achieve the result shown in Figure 5b in the following way. Note that

the figure shows the output when we have not selected either alternative, meaning both are shown using a small multiples approach.

```
MAPTYPE<Overlay[ $\text{bar}_1 \text{ 'alpha' } 0.5, \text{map log bar}_1$ ]  
Overlay[ $\text{bar}_1 \text{ 'alpha' } 0.5, \text{map sqrt bar}_1$ ]
```

Here we are using the *map* function to apply a transformation directly to the visualization elements rather than first transforming the data and only then creating a new visualization.

Finally, we can also perform computations across entire visualizations, such as when sorting elements. Perhaps, for example, we have created some donut charts and realize now that they may be easier to read when sorted. Again, we can avoid having to copy and paste code or start from scratch by directly sorting the elements of an existing visualization.

```
SORTED<Polar (Above [ $\text{pie}_1, \text{pie}_2$ ]),  
Polar (Above [ $\text{sort pie}_1, \text{sort pie}_2$ ]))
```

Figure 5c shows the result.

V. EVALUATION OF VARIATION FOR COMPARISON

To evaluate how well variation and parameterization is able to serve as a model for comparative visualization, we need to know what features are required. Gleicher et al. [12] propose a taxonomy of visual designs used for such comparison tasks. The taxonomy is validated through a significant survey of work.

Their taxonomy of comparative designs categorizes all of the work surveyed into three main categories (as well as pairs of categories) *juxtaposition*, *superposition*, and *explicit representation of the relationships*. Additionally, there are hybrid categories which combine two of these approaches into one design. We explore each of these options in turn and demonstrate which parts of our model can be used to express them.

A. Juxtaposition

The core idea of juxtaposition is to support comparison tasks by placing the objects to be compared into separate spaces. The objects are always shown independently and in their entirety. One common form is spatial juxtaposition, also often called small multiples, in which the objects to be compared are all shown and arranged (often as a grid) in the available space. The taxonomy also allows for juxtaposition in time, in which objects are displayed one after another in sequence.

Our model of variational visualizations supports juxtaposition in more than one way. The easiest way to achieve it is to use variation to encode the visualizations we want to compare, and then rely on the default behavior of our prototype tool. This renders a small multiples view of all visualization alternatives that are not explicitly selected. Another approach is to use the spatial composition operators explicitly, such as **Above** and **NextTo**. These juxtapose visualizations geometrically by dividing up the available space equally. However, plain spatial composition does not support the selective display and navigation of alternatives provided by variations.

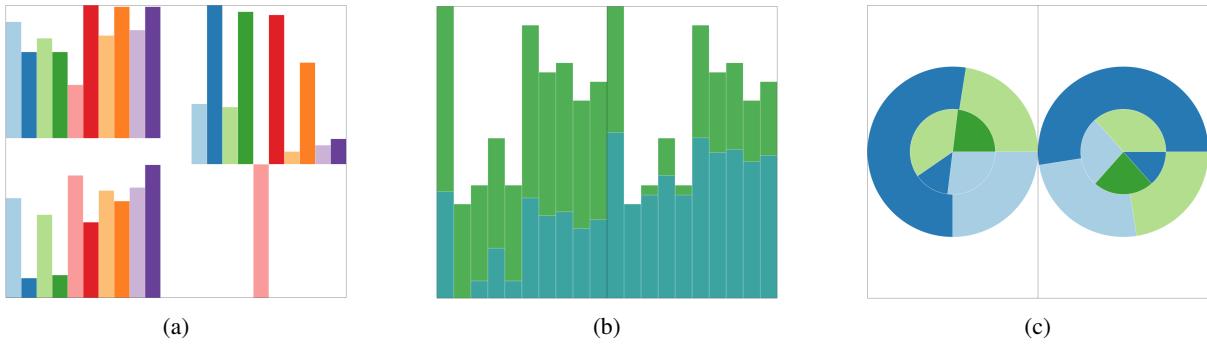


Fig. 5: Examples of comparative visualizations using hybrid designs. In (a) we have two charts on the left which are zipped together to produce the chart on the right by subtracting the heights of the lower chart from the top one. Note that each chart is scaled independently and so simply measuring the bars would be misleading. Figure (b) shows a small multiples rendering of a data set overlaid with its log transformed data on the left and square root transformed data on the right. Finally, (c) shows a variational visualization in which the left variant shows the original data and the right variant shows the visualizations sorted after their creation.

Finally, juxtaposition can also be temporal, as in an animation that cycles through a set of visualizations. Our prototype does not support animation directly, but it is trivial to replicate this behavior using choices. We simply encode the variants as part of a variational visualization, as in the first example, and map each step of the animation to a particular selection. It is then easy to envision a tool which allows the user to define an animation by setting a timer which navigates among the desired selections.

Therefore, we can say that juxtaposition can be modeled by a combination of variation and spatial composition.

$$\text{Juxtaposition} \approx V \oplus \text{SpatialComposition}$$

B. Superposition

The superposition category includes designs in which the objects being compared all share a single space. In general, this is realized by composing visualizations via an overlay operation. Aesthetic tweaks such as transparency and small shifts to avoid totally obscuring some objects are common. Superposition also frequently requires some computation to determine an alignment for different objects. In Section IV-C we showed examples making use of overlays, transparency, and spacing directly.

Because our model offers the ability to define functions over visualizations, realizing a general mechanism for parameterization, we can employ computation at essentially all levels. We have shown examples that include sorting values. We can also envision more sophisticated scenarios such as changing the order of overlaid charts or organizing a small multiples layout based on some derived value from a set of charts.

It is clear from these examples that superposition can be modeled by parameterization/computation and **Overlay**.

$$\text{Superposition} \approx \text{Parameterization} \oplus \text{Overlay}$$

C. Explicit Representation of Relationships

The final category of the taxonomy includes designs which encode the relationships among the objects being visualized directly. One example we have already seen of this is charting

the difference between two ordered data sets (using *zipWith*) and visualizing that result rather than showing both original data sets. A design in this category always involves the extra step of computing the relationships among objects before anything can be visualized. In general, visualizations following these design principles do not require variation techniques in our model, since we have access to computation. We have also shown how we can apply data transformations to individual visualizations, including log and square root transformations.

Finally, in some cases variation can also be used to explicitly encode relationships. One example of this is the pie chart in Section IV-C in which the variation controls the visible level of detail.

Based on these examples, we see that the explicit encoding approach can be modeled by a combination of variation and parameterization.

$$\text{ExplicitEncoding} \approx \text{Parameterization} \oplus V$$

D. Hybrid Categories

The taxonomy also includes designs that take hybrid approaches. Combining the three original categories into pairs results in three new hybrid categories, juxtaposition and superposition, juxtaposition and explicit encoding, and superposition and explicit encoding. Each of these is manifested in designs included in the survey and so are necessary to include.

Due to our compositional design of visualizations, all of the functionality discussed so far is essentially orthogonal and all techniques can be composed. For example, to support juxtaposition and superposition at the same time, we can use any of the approaches mentioned above in Section V-B to produce visualizations making use of superposition. With those results, we can then compose those charts together (using language constructs or variation, as described in Section V-A) to produce a hybrid visualization.

Analogously, for juxtaposition and explicit encoding we can apply any desired computations to explicitly visualize relationships among objects and data and then compose

those into larger, hybrid visualizations. One example of this would be to juxtapose (using variation) two charts which are themselves variational, as we did with the log and square root transformation example in Section IV-C.

Finally, for a hybrid approach involving superposition and explicit encoding we can compute any desired relationships among objects and then add them to the overlay composition used in superposition. Conveniently, the same log and square root transformation example also demonstrates an example of this category.

E. Evaluation Conclusions

Since our model is intentionally limited to a small subset of visualization types we do not claim to be able to reproduce most of the actual designs surveyed to produce the taxonomy. However, we have shown how an approach based on parameterization and variation can, in principle, support any combination of identified comparative designs (see the summary in the following table).

	<i>V</i>	<i>Parameterization</i>	<i>Spatial Composition</i>	<i>Overlay</i>
Juxtaposition	×		×	
Superposition		×		×
Explicit Repr.	×	×		

Our prototype implementation is able to handle all of the core ideas underlying the comparative designs.

VI. RELATED WORK

There is no shortage of visualization tools and models, and it is beyond the scope of this section to characterize them all. We therefore mention only on those which directly influenced this work. While D3 [13] has since supplanted it and gained widespread adoption, its predecessor Protovis [14] is closer in design to our model. Protovis was based on a declarative domain-specific language which separated the specification of visualizations from the rendering process [15]. Protovis was superseded by D3 primarily because the authors aspired to create a tool for web developers to be able to do more than creating visualizations. D3 is less a visualization tool than a library for data-driven web content. The cost of this added flexibility is the elimination of domain-specific constructs. The change seems to have been motivated by a rethinking of the goals of the project. We believe that the domain-specific approach still has value for many users, which is witnessed to some extent by the creation of many libraries in the community that abstract over D3.

Both ggplot2 [16] and the grammar of graphics which underpins it [17] serve as inspiration particularly for visualization transformations, although it is not designed to support programmability and is therefore generally fixed in what operations are supported.

The Haskell domain-specific language Diagrams, described partially by Yorgey [18], supports the creation of diagrams through composition and relatively spacing and directly informs

many of the concepts used to define our model of visualizations. Diagrams does not directly support data-driven graphics and so is not suitable for a general purpose visualization tool.

Comparative visualization is a large and growing field. Over roughly the past two decades, a number of works in information and scientific visualization have advocated for and distinguished deliberate visual comparison designs, including Pagendarm and Post [19], Woodring and Shen [20], and Roberts [21]. Naturally Gleicher et al. [12], referred to throughout this work, provides a thorough overview of the field as well as a taxonomy of comparative designs.

Comparison tasks are also a core part of visualization history tools. Interacting with the history of a user-created visualization artifact is itself too broad of a subject to fully summarize here, so we refer to Heer et al. [2] which studies and organizes the design space of graphical history tools.

Finally, another area in which visual comparison tasks arrive routinely is in uncertainty visualization. Uncertain or missing data often lead naturally to a large, or even infinite set of possible visual representations. One example is weather forecasting with uncertain parameters, which can result in needing to compare a collection of different results, as described by Bonneau et al. [3]. That work also effectively summarizes sources of uncertain data as well as current approaches and unsolved problems.

To our knowledge, no work has yet tried to apply a systematic model of variation explicitly to support visual comparison tasks. However, some work makes implicit use of comparison for variation-based exploratory tasks. For example, Side Views [22] and Parallel Paths [23] designed live “what-if” previews for graphical operations which implicitly relies on comparison. Hartmann et al. [24] took a variation-based approach to user interfaces and interactions which require comparison tasks. As mentioned, the work on variational pictures [9] makes use of variational area trees to help support comparison and exploration tasks.

VII. CONCLUSIONS & FUTURE WORK

We have shown an approach to information visualization based on parameterization and variability. Through examples, we have demonstrated the suitability of this approach for creating not only visualizations in general, but specifically those that support visual comparison tasks.

We have evaluated our model by showing how it is able to instantiate visualizations in every category of Gleicher’s taxonomy of comparative designs [12]. Accordingly, we posit that parameterized variational visualizations offer an effective model of comparative visualization more generally.

In future work, we will extend the implementation of our visualization DSL with general control structures and operators to introduce, maintain, and consume variational visualizations. This will offer users an exploratory approach to information visualization in general.

ACKNOWLEDGEMENTS

This work is partially supported by the National Science Foundation under the grants IIS-1314384 and CCF-1717300.

REFERENCES

- [1] D. Reinsel, J. Gantz, and J. Rynding, “Data age 2025: The evolution of data to life-critical,” IDC, Tech. Rep., 2017.
- [2] J. Heer, J. Mackinlay, C. Stolte, and M. Agrawala, “Graphical histories for visualization: Supporting analysis, communication, and evaluation,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 6, pp. 1189–1196, 2008.
- [3] G.-P. Bonneau, H.-C. Hege, C. R. Johnson, M. M. Oliveira, K. Potter, P. Rheingans, and T. Schultz, “Overview and state-of-the-art of uncertainty visualization,” in *Scientific Visualization: Uncertainty, Multifield, Biomedical, and Scalable Visualization*, C. D. Hansen, M. Chen, C. R. Johnson, A. E. Kaufman, and H. Hagen, Eds. Springer London, 2014, pp. 3–27.
- [4] D. Holten and J. J. van Wijk, “Visual comparison of hierarchically organized data,” in *Joint Eurographics / IEEE VGTC Conference on Visualization*, ser. EuroVis ’08, 2008, pp. 759–766.
- [5] E. R. Tufte, *The Visual Display of Quantitative Information*, 2nd ed. Graphics Press LLC, 2001.
- [6] K. Smeltzer, M. Erwig, and R. Metoyer, “A transformational approach to data visualization,” in *International Conference on Generative Programming: Concepts and Experiences*, 2014, pp. 53–62.
- [7] J. Bertin, *Semiology of Graphics: Diagrams, Networks, Maps*. Morgan Kaufmann Publishers Inc., 1999, English translation.
- [8] M. Erwig and E. Walkingshaw, “The choice calculus: A representation for software variation,” *ACM Transactions on Software Engineering and Methodology*, vol. 21, no. 1, pp. 6:1–6:27, 2011.
- [9] M. Erwig and K. Smeltzer, “Variational Pictures,” in *Int. Conf. on the Theory and Application of Diagrams*, ser. LNAI 10871, 2018, pp. 55–70.
- [10] K. Smeltzer and M. Erwig, “Variational lists: Comparisons and design guidelines,” in *ACM SIGPLAN International Workshop on Feature-Oriented Software Development*, 2017, pp. 31–40.
- [11] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, “An analysis of the variability in forty preprocessor-based software product lines,” in *ACM/IEEE International Conference on Software Engineering*, 2010, pp. 105–114.
- [12] M. Gleicher, D. Albers, R. Walker, I. Jusufi, C. D. Hansen, and J. C. Roberts, “Visual comparison for information visualization,” *Information Visualization*, vol. 10, no. 4, pp. 289–309, 2011.
- [13] M. Bostock, V. Ogievetsky, and J. Heer, “ \mathbb{D}^3 : Data-driven documents,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 2301–2309, 2011.
- [14] M. Bostock and J. Heer, “Protovis: A graphical toolkit for visualization,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1121–1128, 2009.
- [15] J. Heer and M. Bostock, “Declarative language design for interactive visualization,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 6, pp. 1149–1156, 2010.
- [16] H. Wickham, *ggplot2: Elegant Graphics for Data Analysis*, 2nd ed. Springer, 2016.
- [17] L. Wilkinson, *The Grammar of Graphics*. Springer Science & Business Media, 2006.
- [18] B. A. Yorgey, “Monoids: Theme and variations (functional pearl),” in *Haskell Symposium*, 2012, pp. 105–116.
- [19] H.-G. Pagendarm and F. H. Post, “Comparative visualization: Approaches and examples,” in *Visualization in Scientific Computing*, M. G. andāĀ Heinrich Müller andāĀ Bodo Urban, Ed. Springer-Verlag, 1995.
- [20] J. Woodring and H.-W. Shen, “Multi-variate, time varying, and comparative visualization with contextual cues,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 909–916, 2006.
- [21] J. C. Roberts, “State of the art: Coordinated multiple views in exploratory visualization,” in *International Conference on Coordinated and Multiple Views in Exploratory Visualization*, 2007, pp. 61–71.
- [22] M. Terry and E. D. Mynatt, “Side views: Persistent, on-demand previews for open-ended tasks,” in *ACM Symp. on User Interface Soft. and Tech.*, 2002, pp. 71–80.
- [23] M. Terry, E. D. Mynatt, K. Nakakoji, and Y. Yamamoto, “Variation in element and action: Supporting simultaneous development of alternative solutions,” in *SIGCHI Conf. on Human Factors in Comp. Systems*, 2004, pp. 711–718.
- [24] B. Hartmann, L. Yu, A. Allison, Y. Yang, and S. R. Klemmer, “Design as exploration: Creating interface alternatives through parallel authoring and runtime tuning,” in *ACM Symposium on User Interface Software and Technology*, 2008, pp. 91–100.

Creating Socio-Technical Patches for Information Foraging: A Requirements Traceability Case Study

Darius Cepulis, Nan Niu

Department of Electrical Engineering and Computer Science

University of Cincinnati

Cincinnati, OH, 45221, USA

cepulide@mail.uc.edu, nan.niu@uc.edu

Abstract—Work in information foraging theory presumes that software developers have a predefined patch of information (e.g., a Java class) within which they conduct a search task. However, not all tasks have easily delineated patches. Requirements traceability, where a developer must traverse a combination of technical artifacts and social structures, is one such task. We examine requirements socio-technical graphs to describe the key relationships that a patch should encode to assist in a requirements traceability task. We then present an algorithm, based on spreading activation, which extracts a relevant set of these relationships as a patch. We test this algorithm in requirements repositories of four open-source software projects. Our results show that applying this algorithm creates useful patches with reduced superfluous information.

Index Terms—Information foraging theory, Spreading activation, Requirements Traceability

I. INTRODUCTION

If we understand how a user seeks information, then we can optimize an information environment to make that information easier to retrieve. Pirolli and Card worked to understand information-seeking by defining information foraging theory [1], [2]. Information foraging theory (IFT) describes a user’s information search by equating it to nature’s optimal foraging theory: in the same way that scent carries a predator to a patch where it may find its prey, a user follows cues in their environment to information patches where they might find their information.

IFT has seen many applications since Pirolli’s seminal work. For example, in web search, foragers follow information scent to their patches, web pages, helping developers design the information environment of their web pages [3]–[5]. In code navigation and debugging, IFT describes how developers seek to resolve a bug report by navigating from fragment to fragment of code to define and fix the problem [6], leading to models which assist in this process [7]. In both of these scenarios, the patch is clearly defined: in web search, a forager’s patch is a web page, and in debugging, a developer’s patch might be a fragment of code. What happens, though, when the patch is not clearly defined?

Consider socio-technical systems, where information artifacts are connected to people. Facebook, YouTube, GitHub, and Wikipedia all have information artifacts, like posts and code snippets, with a rich context of social interactions tying

them together. A forager simultaneously traverses both the artifacts and the social structures behind them in an information seeking task, making a patch difficult to define. In this paper, we describe a method for delineating patches in such environments.

Requirements traceability is an ideal field for examining patch creation in a socio-technical environment. Requirements traceability is a socio-technical system used to describe and follow the life of a requirement by examining the trail of artifacts and people behind them, from the requirement’s inception to implementation. With a traceability failure, US Food and Drug Administration might cast doubt in product safety [8], or the CEO of a prominent social media company cannot explain to Congress how a decision to withhold information from customers was made [9]. In Gotel and Finkelstein’s words [10], problems like these arise when questions about the production and refinement of requirements cannot be answered. Applying information foraging to these traceability questions could significantly increase efficiency and efficacy of these traceability tasks. In foraging terms, if questions represent prey, what represents a patch?

This paper makes two contributions by deriving a method for delineating these patches. First, by examining requirements socio-technical graphs constructed from four requirements repositories, containing 111 traceability questions, we identify classes of relationships that should be considered in similar requirements traceability tasks. Second, we derive an algorithm, based on spreading activation, which combines these classes with information foraging concepts to create relevant patches where foragers can conduct their traceability tasks. The patches that our algorithm produces are as small as 5–10 nodes representing knowledgeable users and useful information artifacts. Our methods for identifying these classes and deriving this algorithm can be extended to other socio-technical tasks.

II. BACKGROUND

The constructs provided by IFT were first used to analyze how a web user might search for information online [3], modeling scent as relatedness of a link to the forager’s prey. This work eventually developed into the Web User Flow by Information Scent (WUFIS) algorithm [4], where the web was modeled as a graph with sites as nodes, links as edges, and

scent as edge weight. With a spreading activation algorithm, each node was assigned with a value which represents the probability that a forager, given their current location and information need, will navigate to a specific page.

This design was utilized to model programmer navigation in the development of Programmer Flow by Information Scent (PFIS) [11] and its subsequent revisions [12], [13]. PFIS built upon the spreading activation of WUFIS by applying it to the field of developer navigation [11], inferring the forager's goal [12], and creating multi-factor models with PFIS [13]. When inferring the forager's goal, PFIS authors introduced the concept of heterogeneity to their network: in addition to linking code fragments, the PFIS algorithm also linked code fragments to key words, creating a more nuanced topology. Inspired by this heterogeneous approach, we take spreading activation to the socio-technical realm.

In order to develop a spreading activation algorithm in the socio-technical realm, we first examine work conducted in socio-technical graphs. In the Codebook [14] project, people and work artifacts were “friends” in a social network. A user might be connected to an email they sent, bug they closed, and a commit they pushed. By using a single data structure to represent these people, artifacts, and relationships, and a single algorithm (regular language reachability) to analyze this graph, Codebook could handle all the inter-team coordination problems identified in a survey responded by 110 Microsoft employees [15], including requirements traceability problems.

Codebook addresses problems by having a project personnel cast their coordination needs into regular expressions. This is a manual task, requiring deep domain expertise. In contrast, spreading activation can provide a mechanism for automated querying. We therefore adopt Codebook's underlying data structure, but instead of regular language reachability, we adapt the people-artifact graph for spreading activation.

III. EXAMINING REQUIREMENTS SOCIO-TECHNICAL GRAPHS

In order to successfully create patches in a requirements traceability environment, we must first understand the characteristics of the environment [16], [17]. To do this, we construct graphs of the environment following the Codebook paradigm. We then examine the types of human-human and human-artifact relationships that connect a traceability question to an identified answer; encoding these relationships can build a patch that a requirements traceability forager can explore to understand their traceability question better.

A. Constructing Requirements Socio-Technical Graphs

Issue trackers are essential for open-source projects to manage requirements [18]–[23]. Although the requirements of an open-source project can originate from emails, how-to guides, and other socially lightweight sources [24], the to-be-implemented requirements “eventually end up as feature requests in an issue-tracking system” [20]. We therefore turn to the issue-tracking system Jira to understand the life of requirements.

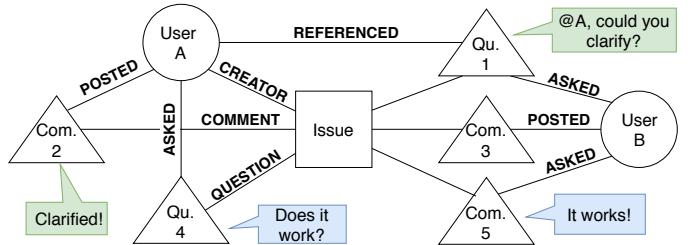


Fig. 1. Requirements Socio-Technical Graph

From Jira, we select four open-source projects from the Apache software foundation [25] or the JBoss family [26]: DASHBUILDER, DROOLS, IMMUTANT, and JBTM. The four projects tackle problems in different domains with implementations written in different programming languages. Within the chosen projects, we focus on questions and answers. By finding the exact person who answered a forager's question, we can gain insight into the information environment. For each project, two researchers identified comments that were questions and identified the respective answer comments, as described in [27].

The next step to creating patches in this network is to build a requirements socio-technical graph (RSTG) so that we can inspect the topology of the network using graph theory concepts. By manually inspecting common interactions, we are able to define the nodes and edges in our network topology. Consider the following example: Figure 1 is a subgraph from the IMMUNANT project, showing two questions and their answers. User A created the issue. User B, the forager, commented on the issue (Qu. 1), asking User A for clarification by referencing them in the comment. User A commented on the issue (Com. 2), providing the clarification. In another foraging interaction, User A, now the forager, commented on the issue (Qu. 4), and User B responded (Com. 5).

Contemporary approaches to requirements traceability are either artifact-based (e.g., trace retrieval) and would consider only the comments and issues [28], or are driven by social roles [29]. However, this interaction and many like it indicated to us that we should represent our environment with comments, issues, and social roles as visible in Figure 1. While some relationships could be considered as unidirectional, e.g., a user posting a comment, the comment also serves as a bridge connecting the issue to the user, who is knowledgeable on the issue. We therefore elected to encode the network as an undirected graph.

B. Properties of Requirements Socio-Technical Graphs in Information Foraging

We now tie our RSTGs to information foraging theory. When a forager asks a question, like the ones in Figure 1, what path might they typically follow to the user who will provide the answer, gathering information on their prey along the way? By analyzing the paths connecting question nodes and answer nodes in the RSTG generated for each question, we identified recurring patterns connecting traceability questions

with answers. The patterns are organized by degrees of socio-technical separation, which we define as the minimum number of edges to be traversed between two nodes.

1) *One or Two Degrees of Separation*: More than three-quarters of answers to our 111 questions are within two degrees of the question. Thirty-one percent of answers were provided by users one degree away: either the asker referenced the user who will answer the question, or the asker themselves answered the question. Recall that when a user is referenced in a comment, they are connected to the comment. Fifty-five percent of answers were provided by users two degrees away: these users were the Creators or Assignees of the issue.

2) *Collaborators and Contributors—Three or Four Degrees*: Eleven percent of our answers came within 3 degrees of separation; most of these answers fell within two classes that we called *Frequent Collaborators* or *Frequent Contributors*. Frequent Collaborators were users who were not connected to the issue, but were connected to the person asking. Frequent Contributors were users who commented or asked questions one or more times on an issue. Users A and B in Figures 1 and 2 are Frequent Contributors, connected by their multiple comments on the issue. The remaining 3% of answers, four degrees away, were Frequent Collaborators of the Creator or Assignee.

3) *Unconnected Users*: The final pattern observed was one instance of an answer unconnected from the graph of the project. At the time the question was asked, the user who will eventually answer the forager’s traceability question was not yet connected to the project by the relationships we chose to express as edges.

It appears, from these classes, that a substantial amount of traceability foraging takes place in four-or-less degrees of separation. Seeking to apply information foraging to traceability, one could simply present all nodes within four degrees of the question as a patch where the forager might seek to understand their question. This would satisfy our requirement of encoding frequently-traversed foraging paths into a patch. However, these patches are extremely large. Including all nodes within 2 degrees of separation produced patches with a mean size of 427 nodes. Including all within 4 degrees produced patches with a mean size of 2186 nodes. With some notion of relatedness, however, these patches could be made smaller without losing relevant information.

IV. CREATING SOCIO-TECHNICAL PATCHES FOR INFORMATION FORAGING

In order to create smaller patches which still contain the described relationships, we turn to the foraging concept of scent, which we define, following WUFIS and PFIS, as the “inferred relatedness of a cue to the prey, as measured by amount of activation from a spreading activation algorithm”. We define “relatedness” in our domain as the amount of knowledge that a user has on an information artifact; this amount is encoded as weight on the edge connecting a user to the artifact. Note that, to support our implementation of spreading activation, a lower weight represents a more powerful connection.

Our strongest connections are Comments/Questions to Issues (weight = 1); they are directly part of the traceability history of an issue. Next, the Creator and Assignee have a high degree of knowledge on a given issue, answering 55% of questions, though an issue can develop without the supervision of the creator or assignee (weight = 2). Next, the user who wrote the comment has determined that the referenced user should have strong knowledge on the comment (weight = 2). Finally, Comments’ and Questions’ connections to users are given the lowest weight (weight = 3, 4) because the user commenting or asking is not guaranteed to have knowledge on the issue to the same degree as Creator or Assignee. Figure 2 shows an RSTG with these weights applied.

With weight defining the relatedness between two given nodes, a given node’s relatedness to the forager’s information need can be determined with spreading activation. Our variant of spreading activation starts at the question-node. The question’s activation is set to 1. Then, surrounding nodes are traversed, and activation is spread from their predecessors. Spreading activation traditionally has each node firing to its successors; our predecessor variant exhibits greater decay while still producing useful networks.

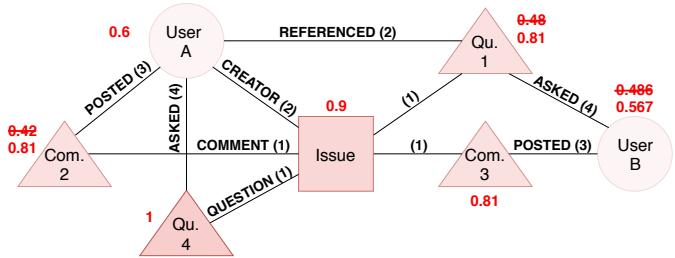


Fig. 2. Spreading Activation Applied to an RSTG (Without Frequency Bonus)

If a node has no activation yet, the activation is simply spread to the new node, decaying more if the weight value is higher. However, if a node already has activation, the higher activation value is spread. This can be seen in Figure 2. Finally, in order to incentivize frequency (in order to promote the Frequent Collaborator and Frequent Contributor patterns), if a node already has activation (i.e., the node has an existing relationship to the question), a percentage of that existing activation is added to the new activation.

By grouping together nodes with high activation, a patch with nodes related by the relationships described previously will be created. To do this, though, a threshold of “high” activation must be defined. Earlier, creating patches by simply enclosing all nodes within four degrees of separation was proposed, because all answers in our dataset fell within four degrees. We now consider those answers’ activations. Examining graphs of the classes discussed earlier, with spreading activation completed, reveals that a forager setting the cutoff at 0.45 would include 100% of results, just like 4 degrees. We could set the cutoff as high as 0.72 and still include 84% of results.

V. RESULTS AND ANALYSIS

Considering just these two extreme cutoffs, we find smaller patches than 4 degrees' mean of 2186 nodes: including all nodes with activation ≥ 0.45 , we have patches of mean size 1281.2, and ≥ 0.72 yields patches of just mean size 7.4. Statistically comparing 4 Degrees and Activation ≥ 0.45 , we conclude that the two sets are non-identical ($t = 10.901$, p-value < 0.01). A similar test for Activation ≥ 0.45 and ≥ 0.72 reaches the same conclusion ($t = 9.6481$ p-value ≥ 0.01). In other words, each cutoff has significantly smaller patches than the previous.

While patch sizes at cutoff activation ≥ 0.45 are still too big for timely foraging, patch sizes at ≥ 0.72 are reasonable. That being said, ≥ 0.72 patches frequently do not include the answer node for three or four degree of separation relationships. However, within these patches, we believe that foragers would still find information relevant to their information need.

Figure 2 serves as a practical example of both the mechanism of the algorithm and a tradeoff to consider. Figure 2 is a network with a cutoff set to ≥ 0.56 —a cutoff chosen for its inclusion of many Frequent Contributors. Indeed, it was a Frequent Contributor that answered User A's question; User B had commented twice on the issue already. While the question ("Does it work?") was a pointed request, asking for a specific piece of information, the patch generated by the request yields not only the user who will answer the request, but also related traceability information.

Figure 2 also demonstrates a limitation of the algorithm in its current state. Other implementations of spreading activation begin from one or more nodes; we could have started the activation from the question *and* the asking user. We chose to include only the question node, as to avoid superfluous information from the asking user's connections. In this case, though, had we included User A as an initial node for activation, the algorithm would have assigned higher activations to the direct collaboration between User B and User A (User A–Question 1–User B). This collaboration was key to the traceability history of the issue.

VI. IMPLICATIONS

Piorkowski and his colleagues [6] codified the fundamental challenges faced by software developers when foraging in the information environment. We believe that our socio-technical approach can help directly address the challenge of "prey in pieces" where the foraging paths were too long and disconnected by different topologies. By explicitly integrating humans in the underlying topology, information foragers can exploit a richer set of relationships.

Codebook [15] confirmed that the small-world phenomenon [30] was readily observed in the socio-technical networks built from the software repositories. Our findings suggested that RSTGs are even smaller with relevant nodes surrounded in four-or-less degrees of separation from the traceability forager's question. Meanwhile, our results revealed several common relationships and their compositions. In light of the recent work on collecting practitioners' natural-language

requirements queries (e.g., [31]–[33]), the patterns uncovered by our study could be used to better classify and answer project stakeholders' traceability needs.

Automated requirements traceability tools have been built predominantly by leveraging text retrieval methods [28]. These tools neglect an important factor—familiarity—which we find plays a crucial role in tracking the life of a requirement. Our results here are to be contrasted with the empirical work carried out by Dekhtyar *et al.* [34] showing that experience had little impact on human analysts' tracing performance. While a developer's overall background may be broad, we feel that the specific knowledge about the subject software system and the latent relationships established with project stakeholders do play a role in requirements tracing. Automated ways of inferring a developer's knowledge degree (e.g., [35]) would be valuable when incorporated in traceability tools.

VII. CONCLUSION AND FUTURE WORK

By considering common relationships connecting a requirements traceability question to its answer, and encoding these relationships into a spreading activation algorithm, we were able to delineate patches for use in understanding context surrounding requirements traceability questions in a socio-technical environment. In this process, we found that traceability questions were answered by users within four degrees of socio-technical separation; these users were typically Frequent Collaborators of the forager or of the creator/assignee of the issue, or Frequent Contributors to the issue. Encoding these relationships as parameters to a spreading activation algorithm resulted in patches of nodes that a forager could traverse, searching for their answer. While simply creating patches including all nodes within four degrees of socio-technical separation would include all answers, the addition of spreading activation created significantly smaller patches.

This method can further be extended within the requirements traceability realm. With only three node types, we were able to generate these patches. With a higher diversity of information, such as code artifacts and commits, or semantic similarity, more nuanced relationships could be determined. Future work could be conducted on the implementation of the algorithm itself, too. Our parameters were set through observation and trial and error. More sophisticated statistical analyses could help better set these parameters. Our method only suggested the first patch for foraging; in reality, a forager will go through several patches in search of their prey. To accommodate this pattern, this work could be extended to multiple-patch creation.

Finally, we determined nodes and edges by thoughtfully examining our environment. While the exact node and edge types would be different, applying the foundation of our design thinking and methods to new domains would create socio-technical graphs where spreading activation can provide relevant and small patches like ours.

ACKNOWLEDGMENT

The work is funded by the U.S. NSF Grant CCF-1350487.

REFERENCES

- [1] P. Pirolli and S. K. Card, "Information foraging in information access environments," in *Conference on Human Factors in Computing Systems (CHI)*, Denver, CO, USA, May 1995, pp. 51–58.
- [2] P. Pirolli, *Information Foraging Theory: Adaptive Interaction with Information*. Oxford University Press, 2007.
- [3] ——, "Computational models of information scent-following in a very large browsable text collection," in *Conference on Human Factors in Computing Systems (CHI)*, Atlanta, GA, USA, March 1997, pp. 3–10.
- [4] E. H. Chi, P. Pirolli, K. Chen, and J. E. Pitkow, "Using information scent to model user information needs and actions and the web," in *Conference on Human Factors in Computing Systems (CHI)*, Seattle, WA, USA, March-April 2001, pp. 490–497.
- [5] X. Jin, N. Niu, and M. Wagner, "Facilitating end-user developers by estimating time cost of foraging a webpage," in *Visual Languages and Human-Centric Computing (VL/HCC), 2017 IEEE Symposium on*. IEEE, 2017, pp. 31–35.
- [6] D. Piorowski, A. Z. Henley, T. Nabi, S. D. Fleming, C. Scaffidi, and M. M. Burnett, "Foraging and navigations, fundamentally: developers' predictions of value and cost," in *International Symposium on Foundations of Software Engineering (FSE)*, Seattle, WA, USA, November 2016, pp. 97–108.
- [7] J. Lawrence, C. Bogart, M. M. Burnett, R. K. E. Bellamy, K. Rector, and S. D. Fleming, "How programmers debug, revisited: an information foraging theory perspective," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 197–215, February 2013.
- [8] P. Mäder, P. L. Jones, Y. Zhang, and J. Cleland-Huang, "Strategic traceability for safety-critical projects," *IEEE Software*, vol. 30, no. 3, pp. 58–66, May/June 2013.
- [9] Q. Forgey and A. E. Weaver, "Key moments from Mark Zuckerberg's senate testimony," <https://www.politico.com/story/2018/04/10/zuckerberg-senate-testimony-facebook-key-moments-512334?cid=apn>, April 2018, accessed: July 2018.
- [10] O. Gotel and A. Finkelstein, "An analysis of the requirements traceability problem," in *International Conference on Requirements Engineering (ICRE)*, Colorado Springs, CO, USA, April 1994, pp. 94–101.
- [11] J. Lawrence, R. K. E. Bellamy, M. M. Burnett, and K. Rector, "Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks," in *Conference on Human Factors in Computing Systems (CHI)*, Florence, Italy, April 2008, pp. 1323–1332.
- [12] J. Lawrence, M. M. Burnett, R. K. E. Bellamy, C. Bogart, and C. Swart, "Reactive information foraging for evolving goals," in *Conference on Human Factors in Computing Systems (CHI)*, Atlanta, GA, USA, April 2010, pp. 25–34.
- [13] D. Piorowski, S. D. Fleming, C. Scaffidi, L. John, C. Bogart, B. E. John, M. M. Burnett, and R. K. E. Bellamy, "Modeling programmer navigation: a head-to-head empirical evaluation of predictive models," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Pittsburgh, PA, USA, September 2011, pp. 109–116.
- [14] A. Begel and R. DeLine, "Codebook: social networking over code," in *International Conference on Software Engineering (ICSE)*, Vancouver, Canada, May 2009, pp. 263–266.
- [15] A. Begel, Y. P. Khoo, and T. Zimmermann, "Codebook: discovering and exploiting relationships in software repositories," in *International Conference on Software Engineering (ICSE)*, Cape Town, South Africa, May 2010, pp. 125–134.
- [16] N. Niu, W. Wang, and A. Gupta, "Gray links in the use of requirements traceability," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 384–395.
- [17] N. Niu, X. Jin, Z. Niu, J.-R. C. Cheng, L. Li, and M. Y. Kataev, "A clustering-based approach to enriching code foraging environment," *IEEE transactions on cybernetics*, vol. 46, no. 9, pp. 1962–1973, 2016.
- [18] T. A. Alspaugh and W. Scacchi, "Ongoing software development without classical requirements," in *International Requirements Engineering Conference (RE)*, Rio de Janeiro, Brazil, July 2013, pp. 165–174.
- [19] N. A. Ernst and G. C. Murphy, "Case studies in just-in-time requirements analysis," in *International Workshop on Empirical Requirements Engineering (EmpiRE)*, Chicago, IL, USA, September 2012, pp. 25–32.
- [20] P. Heck and A. Zaidman, "An analysis of requirements evolution in open source projects: recommendations for issue trackers," in *International Workshop on Principles of Software Evolution (IWPE)*, Saint Petersburg, Russia, August 2013, pp. 43–52.
- [21] E. Knauss, D. Damian, J. Cleland-Huang, and R. Helms, "Patterns of continuous requirements clarification," *Requirements Engineering*, vol. 20, no. 4, pp. 383–403, November 2015.
- [22] P. Rempel and P. Mäder, "Preventing defects: the impact of requirements traceability completeness on software quality," *IEEE Transactions on Software Engineering*, vol. 43, no. 8, pp. 777–797, August 2017.
- [23] N. Niu, T. Bhowmik, H. Liu, and Z. Niu, "Traceability-enabled refactoring for managing just-in-time requirements," in *Requirements Engineering Conference (RE), 2014 IEEE 22nd International*. IEEE, 2014, pp. 133–142.
- [24] W. Scacchi, "Understanding the requirements for developing open source software systems," *IET Software*, vol. 149, no. 1, pp. 24–39, February 2002.
- [25] "Apache software foundation," <http://www.apache.org>, accessed: July 2018.
- [26] "JBoss family of lightweight cloud-friendly enterprise-grade products," <http://www.jboss.org>, accessed: July 2018.
- [27] A. Gupta, W. Wang, N. Niu, and J. Savolainen, "Answering the requirements traceability questions," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*. ACM, 2018, pp. 444–445.
- [28] M. Borg, P. Runeson, and A. Ardö, "Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1565–1616, December 2014.
- [29] O. Gotel and A. Finkelstein, "Contribution structures," in *International Symposium on Requirements Engineering (RE)*, York, UK, March 1995, pp. 100–107.
- [30] D. Chakrabarti and C. Faloutsos, "Graph mining: laws, generators, and algorithms," *ACM Computing Surveys*, vol. 38, no. 1, pp. Article 2, March 2006.
- [31] P. Pruski, S. Lohar, W. Goss, A. Rasin, and J. Cleland-Huang, "TiQi: answering unstructured natural language trace queries," *Requirements Engineering*, vol. 20, no. 3, pp. 215–232, September 2015.
- [32] S. Lohar, "Supporting natural language queries across the requirements engineering process," in *International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ) Doctoral Symposium*, Gothenburg, Sweden, March 2016.
- [33] S. Malviya, M. Vierhauser, J. Cleland-Huang, and S. Ghaisas, "What questions do requirements engineers ask?" in *International Requirements Engineering Conference (RE)*, Lisbon, Portugal, September 2017, pp. 100–109.
- [34] A. Dekhtyar, O. Dekhtyar, J. Holden, J. H. Hayes, D. Cuddeback, and W.-K. Kong, "On human analyst performance in assisted requirements tracing: statistical analysis," in *International Requirements Engineering Conference (RE)*, Trento, Italy, August-September 2011, pp. 111–120.
- [35] T. Fritz, G. C. Murphy, E. R. Murphy-Hill, J. Ou, and E. Hill, "Degree-of-knowledge: modeling a developer's knowledge of code," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 2, Article 14, March 2014.

Semi-Automating (or not) a Socio-Technical Method for Socio-Technical Systems

Christopher Mendez, Zoe Steine Hanson, Alannah Oleson, Amber Horvath,
Charles Hill, Claudia Hilderbrand, Anita Sarma, Margaret Burnett

Oregon State University
Corvallis, Oregon, USA 97330

{mendezc,steinehz,olesona,horvatha,hillc,minic,anita.sarma,burnett}@oregonstate.edu

Abstract—How can we support software professionals who want to build human-adaptive sociotechnical systems? Building such systems requires skills some developers may lack, such as applying human-centric concepts to the software they develop and/or mentally modeling other people. Effective socio-technical methods exist to help, but most are manual and cognitively burdensome. In this paper, we investigate ways semi-automating a socio-technical method might help, using as our lens GenderMag, a method that requires people to mentally model people with genders different from their own. Toward this end, we created the GenderMag Recorder’s Assistant, a semi-automated visual tool, and conducted a small field study and a 92-participant controlled study. Results of our investigation revealed ways the tool helped with cognitive load and ways it did not; unforeseen advantages of the tool in increasing participants’ engagement with the method; and a few unforeseen advantages of the manual approach as well.

Keywords—GenderMag, gender inclusiveness, socio-technical

I. INTRODUCTION

How should software professionals go about building human-adaptive socio-technical systems? Because socio-technical systems are systems in which humans are intrinsic parts of the system, building such systems effectively requires (1) human-centric concepts, in part (2) to model human behavior—but some developers may not have these skills.

A spectrum of methods—which are themselves socio-technical—exist to help, by integrating (1) and (2) into the design and/or implementation phases of building such systems. Examples include Heuristic Evaluation [43], Cognitive Walkthroughs [37, 58, 60], personas [2, 25, 31], and GenderMag [12]. Teams of software professionals can work together using these socio-technical methods to evaluate socio-technical systems in the design and/or implementation phases of building such systems.

However, methods like these are cognitively heavy, requiring software developers to immerse themselves in perspectives of people different from themselves. This is especially cognitively difficult for modeling people *very* different from themselves—such as having a different gender, as is the case when using the GenderMag method [12, 30].

This raises the question of whether semi-automating such a method might ease developers’ cognitive burden. To investigate this question, we built a Chrome-based web extension for GenderMag called the GenderMag Recorder’s Assistant. The tool semi-automates evaluating any prototype/mockup viewable in a Chrome browser: e.g., web-based apps (mobile or desktop), html mockups, etc.

To use the Recorder’s Assistant, a software team navigates via the browser to the app or mockup they want to evaluate, then starts the tool from the browser menu. The main sequence is to view a persona (Fig. 1(c)) and proceed through the scenario of their choice from the persona’s perspective, one action at a time. At each step, the tool’s “context-specific capture” captures screenshots about the action the team selects (Fig. 1(a)), and records the answers to questions about it (Fig. 1(b)). The tool saves this sequence of screenshots and questions/answers to form a gender-bias “bug report.”

Through these mechanisms, the Recorder’s Assistant aims to reduce the cognitive load for software professionals working with GenderMag in three ways: visually marking the user action that software professionals are currently considering (Fig. 1 (a), box around the action “click on shift”); guiding the software professionals through the GenderMag questions, including a checklist of the persona’s facets to be considered (Fig. 1(b)); and keeping the software practitioners’ chosen persona visible and quickly accessible (Fig. 1(c)).

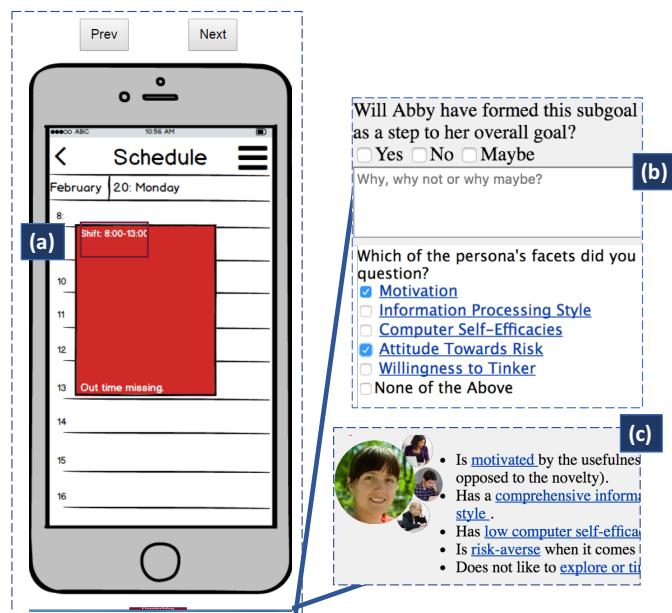


Fig. 1: The Recorder’s Assistant tool during an evaluation of a mobile time-and-scheduling app. (Left): The app being evaluated is displayed with (a) a rectangle around the action the evaluators are deciding if a user like “Abby” will take. (Right): A blow-up of portions of the GenderMag features for the app: (b) the GenderMag question the team is answering at the moment, including a checklist of Abby’s facets; and (c) a summary of the persona the team has decided to use (in this case, Abby).

This work was supported in part by NSF 1314384 and 1528061.

But might a semi-automated tool like the Recorder’s Assistant do more harm than good? One potential problem might be disengagement. That is, since only one member of the software team would actually navigate through the tool, the rest of their team might disengage and become distracted by other apps on their computers (e.g., email and messages). Another might be a decrease in accuracy, such as if the team starts checking off boxes (e.g., Fig. 1(b)) without thinking much about them, or becomes distracted by having to deal with the tool itself.

To investigate whether these issues would arise, we conducted two studies: a small field study at a technology company, and a mixed-methods laboratory study with 92 participants. The following research questions guided our investigation:

- *RQ-Cognitive*: What benefits and disadvantages can a tool like the Recorder’s Assistant bring to software teams’ *cognitive load and recording accuracy*?
- *RQ-Engagement*: Can such a tool manage to “do no harm” to software teams’ *engagement*?

II. BACKGROUND AND RELATED WORK

A. Background: The GenderMag Method

GenderMag [12] is a socio-technical method. Its “socio-” aspect is that a software team works together to use it. Its “technical” aspect is, of course, that what the team is using it for is to evaluate software. It integrates human-centric concepts and mentally modeling other people into the process of evaluating software as follows.

GenderMag’s foundations lie in research on how people’s individual problem-solving strategies sometimes cluster by gender. GenderMag focuses on five facets of problem-solving:

(1) *Motivations*: More women than men are motivated to use technology for what it helps them accomplish, whereas more men than women are motivated by their interest in technology itself [3, 8, 10, 16, 27, 32, 36, 38, 56]. (2) *Information processing styles*: Problem-solving with software often requires information gathering, and more women than men gather information comprehensively—gathering fairly complete information before proceeding—but more men than women use selective styles—following the first promising information, then backtracking if needed [14, 20, 41, 42, 49]. (3) *Computer self-efficacy*: Women often have lower computer self-efficacy (confidence) than their peers, and this can affect their behavior with technology [3, 4, 5, 8, 10, 24, 29, 33, 38, 44, 46, 57]. (4) *Risk aversion*: Women tend statistically to be more risk-averse than men [18, 23, 59], and risk aversion can impact users’ decisions as to which feature sets to use. (5) *Styles of Learning Technology*: Women are statistically more likely to prefer learning software features in process-oriented ways, and less likely than men to prefer learning new software features by playfully experimenting (“tinkering”) [5, 8, 15, 17, 32, 51]. Any of these differences in cognitive styles is at a disadvantage when not supported by the software.

GenderMag brings these facets to life with a set of four faceted personas—“Abby”, “Pat(ricia)”, “Pat(rick)” and “Tim” (Fig. 2). Each persona’s mission is to represent a subset of a system’s target users as they relate to these five facets.

GenderMag intertwines these personas with a specialized Cognitive Walkthrough (CW) [58, 60]. The CW is a long-standing inspection method for identifying usability issues for new users to a program or feature. In a GenderMag CW, evaluators answer a question about each subgoal one of the personas might have in a detailed use-case, and two CW questions about each action, using the persona’s five facets. Further, because GenderMag specializes in inclusiveness, a GenderMag CW inclusively collects answers from multiple team members. The questions are:

SubgoalQ: Will <persona> have formed this subgoal as a step to their overall goal? (Yes/no/maybe, why)

ActionQ1: Will <persona> know what to do at this step? (Yes/no/maybe, why)

Action Q2: If <persona> does the right thing, will s/he know s/he did the right thing & is making progress toward their goal? (Yes/no/maybe, why)

The GenderMag Recorder’s Assistant tool aims to facilitate the recording of these answers.

B. Background: Mentally Modeling People

The GenderMag method’s effectiveness rests on enabling software professionals to mentally *model* other people, a capability called “Theory of Mind.” Theory of Mind is cognitive perspective-taking: the innate human ability to reason and make inferences about another’s feelings, desires, intentions, and goals [47, 53]. Theory of Mind is similar to empathy—but empathy is *emotional* perspective-taking, whereas Theory of Mind is *cognitive* perspective-taking,

An example of Theory of Mind is someone (say, a software developer) building a model in their brain of another person (say, a user) who is different from themselves, and then “executing” that model in a new situation to predict how that person will behave. GenderMag’s personas are meant to facilitate developers’ Theory of Mind modeling of their users.

C. Related Work

GenderMag as a method (unsupported by a tool) has had several evaluations. Marsden and Haag did an eye-tracking study on the GenderMag personas and found that people’s understanding and recollection of the facets were not significantly affected

Abby Jones¹

- 28 years old
- Employed as an Accountant
- Lives in Cardiff, Wales

Background and skills
Abby works as an accountant. She is comfortable with the technologies she uses regularly, but she just moved to this employer 1 week ago, and their software systems are new to her.
Abby says she's a "numbers person", but she has never taken any computer programming or IT systems classes. She likes Math and knows how to think with numbers. She writes and edits spreadsheet formulas in her work.
In her free time, she also enjoys working with numbers and logic. She especially likes working out puzzles like Sudoku.

Motivations and Attitudes

- Motivations: Abby is motivated by the results of using unfamiliar technologies if it makes her life easier. She prefers to use methods and comfortable with tasks she cares about.

Attitude toward Risk: Abby's life is a little complicated and she rarely has spare time. So she is risk averse about using unfamiliar technologies that might need her to spend extra time on them, even if the new features might be relevant. She instead performs tasks using familiar features, because they're more predictable about what she will get from them and how much time they will take.

How Abby Works with Information and Learning

- **Information Processing Style:** Abby tends towards a comprehensive information processing style when she needs to more information. So, instead of acting upon the first option that seems promising, she gathers information comprehensively to try to form a complete understanding of the problem before trying to solve it. Thus, her style is "bursty": first she reads a lot, then she acts on it in a batch of activity.
- **Learning: by Process vs. by Tinkering:** When learning new technology, Abby leans toward process-oriented learning, e.g., tutorials, step-by-step processes, wizards, online how-to videos, etc. She doesn't particularly like learning by tinkering with software (i.e., just trying out new features or commands to see what they do), but when she does tinker, it has positive effects on her understanding of the software.

¹Abby represents users with motivation abilities and information learning styles similar to hers. For data on females and males similar to and different from Abby, see <http://userscouncil.org/gender/gender.php>

Fig. 2. Abby is a “multi-persona”, meaning that she has multiple appearances and demographic portions of her are customizable [31]. One of the facets is blown up for legibility.

by the persona's picture (a favorable finding for these personas), but that people's perceptions of the persona's competence were affected by the picture (an unfavorable finding for these personas) [39]. A follow-up study investigated ways to mitigate this phenomenon, and found that "multi-personas"—in which a single persona shows pictures of different people the persona can represent—helped discourage gender stereotyping [31].

Evaluations of GenderMag's validity and effectiveness have produced strong results. In a lab study, professional UX researchers were able to successfully apply GenderMag, and over 90% of the issues it revealed were validated by other empirical results or field observations, with 81% aligned with gender distributions of those data [12]. In a field study using GenderMag in 2-to-3-hour sessions at several industrial sites [11, 30], software teams analyzed their own software, and found gender-inclusiveness issues in 25% of the features they evaluated. GenderMag has also been used to evaluate a Digital Library interface [21] and a learning management system [55], uncovering significant usability issues in both. In Open Source Software (OSS) settings, OSS professionals used GenderMag to evaluate OSS tools and infrastructure and found gender-inclusiveness issues in 32% of the use-case steps they considered [40]. Finally, in a longitudinal study at Microsoft, variants of GenderMag were used to improve at least 12 teams' products [9].

There is also related work on problems and/or tools on related methods, such as personas and cognitive walkthroughs. Personas were created and developed by Cooper as a way to channel, clarify, and understand a user's goals and needs [19]. Among the benefits claimed from using personas are inducing empathy towards users [2] and facilitating communication about design choices [48]. However, personas are not uncontroversial. Most pertinent to this paper is the issue of personas being ignored. For example, Friess reported that personas were referenced only 2% of the time in conversations regarding product decisions [25]. Friess also found that, even when evaluators used personas alongside CWs as focal points [25, 35], the personas were used only 10% of the time [25]. Thus, in this paper we measure engagement with the personas for both the tool and the paper method.

Regarding problems and tools for the other component of GenderMag, a specialized CW, Mahatody et al.'s [37] comprehensive literature survey of cognitive walkthroughs describes many CW variations, some of which focus on reducing problems with the classic CW [26, 54, 58] such as by reducing the time it requires. Niels et al. recommended that a tool for CWs might address issues like these by guiding the analyst through each CW step, in order to avoid missing steps and to more accurately record results, and to integrate a CW tool into a prototyping tool [34]. (The GenderMag Recorder's Assistant tool follows these recommendations.)

There is only a little work on creating such CW tools, but early in the lifetime of CWs, Rieman et al. created a tool with similar goals as the GenderMag Recorder's Assistant, in that it records the results of a human-run CW [50]. Their study found that analysts' predictions using the tool were accurate. However, their tool was based on an older, much more complex version of the CW, and was a stand-alone recorder, whereas the GenderMag Recorder's Assistant is integrated with the prototype being

evaluated. Most pertinent to this paper, use of their tool was not compared to using a manual/paper version of the CW.

At the other end of the automation spectrum, a few researchers have created tools to automatically perform subsets of the cognitive walkthrough (e.g., [6, 7, 22]). Tools like these are different from the GenderMag Recorder's Assistant in that they handle only subsets of CWs, and are intended to *replace* humans in using such methods, whereas our investigation considers how to *support* humans using such methods. None of these works evaluates how using a tool impacts evaluators' effectiveness when software teams use a socio-technical method like GenderMag. That is the gap this paper aims to help fill.

III. STUDY #1: INITIAL FIELD STUDY

We began with a small field study to gain a real-world perspective. Two professional software developers at a West-Coast technology company, one man and one woman, conducted a GenderMag evaluation of one of their company's mobile printing apps (Fig. 3(left)) using the Recorder's Assistant tool. There are three roles in the process: *facilitator* (runs the walkthrough), *recorder* (records the results), and *evaluator* (answers the questions). One of the developers acted as both the facilitator and recorder, and both developers served as evaluators. We observed and video-recorded the session, which lasted about two hours. Both of the developers had prior experience using the (paper-based) GenderMag method.

Study #1 revealed evidence both against and for the tool reducing cognitive load. On the negative side, the tool sometimes distracted the participants from their evaluation task, essentially stealing cognitive cycles to think about the tool instead of the task when subtleties arose. For example:

West1 (minute 1:28): "ok, perform it...Ummmm, ok, what happened?"
Researcher: "is it not letting you...oh here, hover over that..." *Discoverers duplicated screen shot had been entered, but it looks exactly the same, so looks like tool didn't respond.*

Even so, the team's overall opinion was positive about the

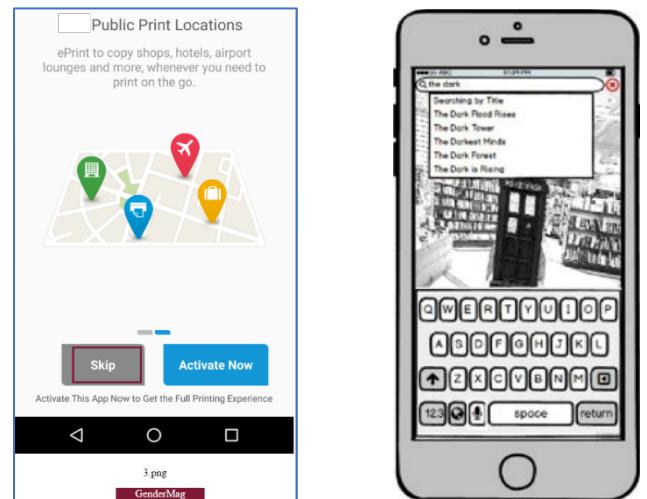


Fig. 3: (Left): A partial screenshot from the field study's mobile printing app; the red rectangle is around the "Skip" actions the participants are currently evaluating. (Right): One project a participant team brought to the controlled study was an "executable" mock-up of an augmented-reality bookstore navigator. Another was Fig. 1's mobile scheduling app.

tool's cognitive benefits. As Participant West1 put it:

West1 (minute 1:48, during debrief, in response to how it compared to paper): "...Way easier. This is way better than the paper version ... It keeps you focused."

For RQ-Engagement, the session also revealed both positive and negative effects. Participant West1 was always engaged—with his/her screen being projected s/he had little choice—but West2 occasionally disengaged from the task, catching up on email instead. Still, the brightly projected image captured both participants' gaze most of the time. More critically, perhaps because it included Abby pictures (as in Fig. 1), their evaluations consistently showed an Abby perspective. For example:

West1 (minute 33): "All indicators encourage the risk-averse user to push the activate button: skip is gray, and the text below ..."

West2 (minute 34): "<if she hits skip> ... <she doesn't> know what's going to happen."

Their faithfulness to an Abby perspective paid off in the kinds of insights Theory-of-Mind methods aim for, such as:

West1 (minute 37): "This <feature> is probably where we're losing half our <target users>."

Interestingly, when the participants needed to think more deeply about Abby, sometimes they chose to study the *paper* version of Abby rather than the version the tool was displaying on the projector—even though the displayed version had explicit links (Fig. 1(c)) to the full details.

West1 (minutes 19-20): "Motivation? Information Processing style?"

Both turn to the paper description and start studying it. West1 reads aloud: "she prefers to use methods she is already familiar and comfortable with..." West1 turns back to screen and marks the facet. West2 (studies paper further): "Maybe information processing style." They both start reading aloud from the paper...

West1 (minute 36, looking at screen): "But Abby would read this, right?" Goes back to studying the paper.

West2: (minute 39): "Even though I've done GenderMag a couple of times, I still have to look at the paper."

West1: (minute 1:42, during debrief, when asked why referred to the paper persona): "I liked the ones up on the screen because they're very succinct... but sometimes I had to go back here <to the paper> because I thought there was something more, some detail that I wanted to consider."

These initial results, which are summarized in Table I, suggested the need for a more in-depth investigation, so we then conducted a controlled, mixed-methods laboratory study.

TABLE I: STRENGTH AND WEAKNESS EVENTS OBSERVED IN THE INITIAL FIELD STUDY.

	RQ-Cognitive	RQ-Engagement
Tool strengths	<ul style="list-style-type: none"> Tool "way easier." 	<ul style="list-style-type: none"> Recorder fully engaged: Tool "keeps you focused."
Tool weaknesses	<ul style="list-style-type: none"> Tool sometimes taxed cognition: e.g., "ok, what happened?" 	<ul style="list-style-type: none"> Non-recorder had laptop open, used it to multi-task. Participants turned to Abby-on-paper, attended less to Abby-in-tool

IV. CONTROLLED STUDY METHODOLOGY

Study #2 used a between-subjects Tool vs. Paper design. We conducted it in two settings at a U.S. university: one setting primarily to collect quantitative data (classroom setting) and the other primarily to collect qualitative data (videorecorded in a lab). In both settings, teams of 2-4 participants performed GenderMag evaluations on their own software

A. Participants (both settings)

The 92 participants were junior and senior students recruited from two computer science courses. These courses enabled a controlled investigation with enough suitable teams for statistical power because: (1) the courses provided a reasonably large pool of software creators already on software teams of similar sizes. (2) These teams were in the process of creating software they cared about for their grades in these courses. (3) Their software was at a stage suitable for a GenderMag evaluation: mature enough to evaluate but early enough that changes could still be made inexpensively.

All students enrolled in the two courses performed the GenderMag evaluations as part of their coursework, but only teams who opted in are part of the reported study. That is, if a team opted into the study, their session outputs became part of our data; otherwise their outputs were used only for the class. Although a few participants had seen or used GenderMag before, their teams did not show any advantage from this: their teams' measures fell near the average (two slightly above, and two slightly below). Participant demographics are shown in Table II.

B. Procedures (both settings)

After the teams were randomly assigned to a treatment, they opted in or not as desired. As Table II shows, this process resulted in about half the participating teams performing their evaluations using the tool, and the rest using the paper materials from the GenderMag kit [13]. As in the field study, participant teams had a real stake in doing these evaluations, because they used the GenderMag method to find problems with their *own* software projects (e.g., Fig. 1(a) and Fig. 3 (right)), which they were developing over the course of the term.

To control variability, we pre-selected which persona—Abby—all teams would use. (If a team wanted to evaluate the software using a second persona, they could do so outside of the study session.) A few days before the sessions, we introduced

TABLE II: PARTICIPATING TEAMS, WITH 2-5 PARTICIPANTS PER TEAM, BY SETTING (COLUMNS) AND BY TREATMENT (ROWS: DARK ROWS ARE TOOL, LIGHT ARE PAPER). TOTALS: 41 TOOL PARTICIPANTS, 51 PAPER PARTICIPANTS.

	Classroom	Video lab	Treatment Totals
Number of teams	10 teams	2 teams	12 teams
	11 teams	3 teams	14 teams
Men	31 men	2 men	33 men
	31 men	7 men	38 men
Women	4 women	2 women	6 women
	9 women	2 women	11 women
Declined to state	0 people	2 people	2 people
	1 people	1 people	2 people
Had seen GenderMag before	4 people	0 people	4 people
	1 people	0 people	1 people

all teams to the Abby persona, and then told them to customize three fields of Abby—her age, place of residence, and occupation—to fit their own software project’s target demographics. For example, GenderMag’s prepackaged Abby is a 28-year-old accountant who lives in Wales, but among the teams’ customizations were Abby as a 16-year-old Oregon high school student and as a 40-year-old Baltimore car mechanic.

We began with a brief tutorial on the GenderMag method. In the Tool treatment, we also helped participants set up the tool on their team’s laptop, and briefly instructed them in how to operate the tool. The teams then performed their GenderMag evaluations, in which they used their customized Abby to walk through a use case they chose in their own software project. At each step, they answered the questions on the CW form (see the Background section) about whether and why Abby would act upon the “right” feature in the way they, the software’s designers, had intended with their design. Finally, each participant filled out a NASA Task Load Index (TLX) questionnaire to report their impressions of cognitive load [28].

C. Treatments (Classroom): GenderMag via Tool vs. Paper

The classroom setting was two large classrooms (one room per treatment), each with multiple teams of 2-4 participants. The Tool teams walked through their use cases with their software prototypes embedded in the tool as in Fig. 1(a), and answered the questions as in Fig. 1(b). The Paper teams did the same things but without a tool: their prototypes were running on their laptops or on paper storyboards, but their CW questions were printed on paper with no limitations on what they could enter (e.g., no checkboxes) and unlimited space. In the Tool treatment, resources (forms, personas, etc.) were primarily computer-based, whereas in the Paper treatment, resources were primarily on paper. However, because some people prefer reading paper over screen and some prefer typing over writing, *both* treatments were allowed to add on use of paper or the computer for reading or writing. For example, some Tool teams turned to paper-based Abby, and some Paper teams typed their CW answers on their laptops using word processing.

D. Treatments (Lab): GenderMag via Tool vs. Paper

Participants in the lab setting followed the same procedures as in the classroom setting, but with their evaluations conducted one team at a time in a lab and videorecorded.

E. Data analysis (both settings)

We combined the settings for analysis. Qualitative data came primarily from the videorecorded setting’s sessions. We transcribed the videos of each session, segmenting the resulting transcripts by conversational turn. We then qualitatively coded each conversational turn for the presence of the number of persona mentions within a conversational turn, any mentions of the persona’s problem-solving facets (e.g., motivations, information processing style, etc.), and the presence of cognitive issues.

To measure how often participants explicitly referred to Abby, we coded each time a participant said “Abby”, “she”, or “her”. To be conservative, we did not count instances of the participant simply reading the CW form questions aloud (“Would Abby have formed this subgoal as a step to her overall goal?”).

We coded instances of facet engagement and of particular

cognitive issues using prior works’ GenderMag code sets for facets and cognitive issues [11, 30] (see the relevant results section for code set details). Two researchers independently coded 20% of the transcripts’ conversational turns using these code sets and obtained 99% agreement (Jaccard index). The two researchers then split up the rest of the coding.

We also coded both settings’ written CW forms for persona mentions and facet mentions using the same code sets as above. We segmented these forms by CW step (i.e., each new CW question started a new segment). Two researchers independently coded 20% of the data and reached 93% agreement (Jaccard index). The two researchers then split up the rest of the coding.

In total, we qualitatively coded 1681 conversational turns from the videorecorded setting and 392 CW form segments from both the videorecorded and classroom settings.

V. RESULTS: CONTROLLED STUDY

A. Results: Cognitive Load and Recording Accuracy

1) Participants’ perceptions of cognitive load

To measure the 92 participants’ perceptions of cognitive load, we used the NASA Task Load Index (TLX) questionnaire [28]. The TLX is a validated questionnaire with six questions, each answered on a scale from 1-21. Four of these questions measure perceived cognitive costs: how hard participants felt they had to work, how rushed the pace of the task was, how stressed they felt during the task’s completion, and how high they felt the mental demand to be. The fifth question measures how successful they felt, and the sixth is on physical exertion.

The results of the participants’ TLX responses were an interesting mix. As Fig. 4 shows, Tool participants felt that they had to work less hard (ANOVA, $F(1,90)=6.14$, $p=.0150$)—but also felt *more* stressed (ANOVA, $F(1,90)=6.4$, $p=.0129$). There were no differences between the two treatments in their perception of physical exertion, the amount of mental demand, or how rushed they felt, but Tool participants felt that they were less successful (ANOVA, $F(1,90)=4.2$, $p=.0445$).

The Tool participants’ perception of working less hard is consistent with the Study #1 comment by participant West1, whose comparison of the tool with their prior experience with

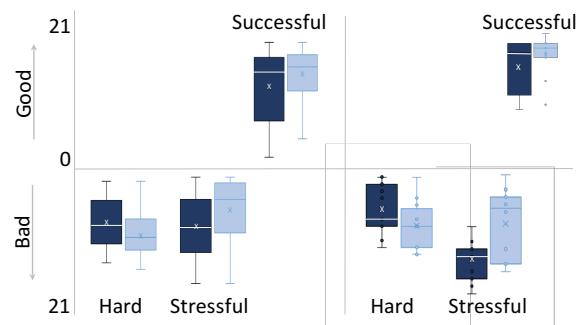


Fig. 4: TLX scores (out of 21). (Left bottom) Tool participants ($N=41$) felt the work was not as hard (down=Harder) as Paper participants did ($N=51$), but felt more “insecure, discouraged, irritated, stressed, annoyed” than Paper participants (down=more Stress). (Left top) Tool participants felt less successful than Paper participants (shown as TLX complement, so Up=more success). (Right) Tool recorders ($N=11$) did not work as hard as Paper recorders ($N=12$), but were much more stressed.

the paper version concluded that the tool was “way easier.” Our interpretation is that the Tool participants’ perception of working less hard was due to the tool keeping them on track when stepping through their prototypes, and also ensuring that their CW answers were tied to the actions they intended those answers for.

The fact that the Tool participants at the same time felt more stressed is also consistent with Study #1 data. Sometimes the tool behaved in ways that participants did not understand or had to be restarted, and this seems likely to have added stress. For example, one participant became confused by the tool’s large collection of prototype screenshots:

Tool2-P1: And after that, oh my god I think there's too many screens.

Stress was particularly high for the Tool teams’ recorders. Tool recorders had a median stress measure of 12, compared to only 5.5 for the Paper recorders. Ultimately, the cognitive cost of the stresses Tool participants reported may have played a part in their perceived lack of success, consistent with Schneider et al.’s findings that cognitive load interferes with Theory-of-Mind effectiveness [53].

2) Actual Recording Accuracy

However, Paper participants’ perceptions of their own success were overly optimistic, or perhaps they simply discounted the importance of recording accuracy. We analyzed the verbalizations in the videorecordings for two types of recording errors: a team discussing a facet and deciding upon it verbally but the recorder omitting it, or the recorder including a facet that the team had not mentioned. The results showed that recording accuracy in both treatments was a bit problematic—but the video-recorded Tool teams recorded their facets more accurately than any of the Paper teams did, with Tool teams averaging 65% accuracy vs. Paper teams averaging only 35% accuracy.

3) Two Cognitive Issues: “Where are we?” & Detours

Prior work has reported accuracy issues in the GenderMag context to be disproportionately tied to two particular cognitive issues: “*Where are we?*” (participants losing track of which action with the prototype they are evaluating), and *detours* (participants digressing from the evaluation, such as getting sidetracked by talking about potential new features for their application) [30]. Thus, following the same procedures as this work, we investigated how often “where are we?” and detours arose for the teams in our study.

The “where are we” problems reported in the prior work rarely occurred, with only 6 instances in total out of a total of 1681 conversational turns, perhaps because the teams were smaller than those experiencing “where are we?” problems in

TABLE III: COGNITIVE LOAD SUMMARY: TOOL VS. PAPER PARTICIPANTS’ PERCEPTIONS OF COGNITIVE LOAD.

	Hard work	Stress	Felt successful
Tool strengths	Not as hard (Study #1 & Study #2)		
Paper strengths		Less stressful (Study #2)	Felt more successful (Study #2)

prior work [30]. However, detours were problematic, with a total of 49 instances spanning over 12% of their conversational turns.

The detours were particularly problematic for Paper teams. As Table IV shows, the videorecorded Tool teams experienced fewer detours than Paper teams—especially lengthy detours. (Since “long” is a matter of judgment, we tried different threshold values, but they reveal similar patterns. Shown are the 5-turn and 10-turn thresholds.) Overall, the greater the number and/or length of detours, the more pervasive the inaccuracy problems. Note in Table IV’s right three columns that, when Tool teams got sidetracked into detours, those teams recovered more quickly and got back on track, consistent with field study participant West1’s comment that the tool “keeps you focused”.

Table V summarizes the results of the Accuracy and Cognitive Issues subsections. Together with the summary of participants’ perceptions of cognitive load (Table III), these results point out that (1) Theory-of-Mind modeling is hard work, and that (2) each of the Tool and the Paper approach have their own strengths in lightening the load.

B. Results: RQ-Engagement

1) Persona Engagement By the Numbers

GenderMag requires real engagement for participants to mentally build and then mentally “execute” models of people not necessarily like them. Thus, to measure engagement, we compared participants’ explicit engagement with Abby (saying/writing “she”, “Abby”, etc.) in three ways: on teams’ written forms, in their verbalizations, and against prior literature.

By all three measures, as Table VI and Fig. 5 (left) summarize, the teams were very engaged with the persona. This was

TABLE IV: THE VIDEORECORDED TEAMS’ ACCURACY PROBLEMS & COGNITIVE ISSUES IN 1681 CONVERSATIONAL TURNS, SORTED BY DEGREE OF INACCURACY (COLUMN 2). GRAY CHANGES AT 15%, 30%, ..., AND HIGHLIGHTS HOW DEGREE OF INACCURACY TENDED TO WORSEN AS DETOURS WORSENNED. PAPER TEAMS TENDED TO HAVE MORE PROBLEMS WITH BOTH.

	Inaccurate recordings (% of facet instances)	Conversational turns spent in Detours + WAWs	“Long” detours (% of detour instances)		
			≥5 turns	≥10 turns	Mean length
Tool2	28%	7%	25%	13%	3.5 turns
Tool1	42%	9%	0%	0%	2.0 turns
Paper1	50%	22%	30%	10%	3.8 turns
Paper3	55%	18%	42%	21%	5.8 turns
Paper2	91%	6%	33%	33%	5.0 turns

TABLE V: SUMMARY OF TOOL VS. PAPER ACCURACY STRENGTHS. RECORDING ACCURACY HAS NO SHADING BECAUSE, ALTHOUGH TOOL WAS MORE ACCURATE THAN PAPER, NEITHER WAS STRONG.

	“Where are we?”	Detours	Recording accuracy
Tool strengths	Few problems (Study #2)	Shorter detours (Study #2)	Better recording accuracy (Study #2)
Paper strengths	Few problems (Study #2)		

true in both treatments: there was no significant difference between the Tool vs. Paper treatment, and both treatments' team engagement with Abby was comparable to prior literature.

However, one surprising similarity in Tool and Paper teams' engagement with Abby was *where* they looked when they wanted to remind themselves of Abby's attributes. Consistent with the Study #1 results, Tool participants often referred back to *paper* versions of Abby. For example:

Tool2-P2: (reads from paper) "Abby uses technology to accomplish her tasks, she learns new technologies when she needs to but prefers to use technology she's already comfortable with." (stops reading): "So yeah. Motivation..."

Tool1-P1: "Um," (reads from paper) "...gathers information to try to form a complete understanding" (stops reading). "Probably none of the above."

An arguably “ideal” level of engagement in a CW-based method like GenderMag would be for a team to refer to Abby at every step in their CW analysis. Remarkably, both the Tool and the Paper teams neared that ideal, referring to Abby in *almost every single segment*: in 94% and 97% of the CW steps, respectively (Table VI, bottom section).

2) Facet Engagement By the Numbers

Recall from Section II that the core of this method lies in its problem-solving facets. Thus, to measure engagement with these facets, we coded each of the teams' written CW forms for mentions of each of the five facets. (Duplicate mentions of the same facet were not counted.) As Fig. 5 (right) shows, the Tool teams mentioned significantly more facets per response than Paper teams did (Fisher's exact test, $p=.0048$, $n=26$).

3) Depth of Engagement

But did the Tool teams mark off checkboxes just because

TABLE VI: ENGAGEMENT: BOTH TOOL AND PAPER TEAMS MENTIONED ABBY AT RATES COMPARABLE TO PRIOR GENDERMAG RESULTS, AND BETTER THAN THE BEST PRIOR NON-GENDERMAG PERSONA RESULTS WE HAVE BEEN ABLE TO LOCATE. PRIOR RESULTS ARE SHADED. TOOL VS. PAPER RESULTS WERE NOT SIGNIFICANTLY DIFFERENT.

Source	Explicitly mentioned persona (Abby)
Prior field work on personas [25]	...verbally in 10% of conversational turns
Prior GenderMag field study (using paper) [30]	...verbally in 23% of conversational turns
Prior GenderMag lab study (using paper) [31]	...verbally in 34% of conversational turns
Tool teams	...verbally in 24% of conversational turns
Paper teams	...verbally 28% of conversational turns
<hr/>	
Prior GenderMag field study (using paper) [30]	...verbally while discussing 79% of the CW steps
Tool teams	...written on 49% of CW steps, and ...verbally in 94% of CW steps
Paper teams	...written on 62% of CW steps, and ...verbally in 97% of CW steps

they were there, without really deciding on them? (Indeed, in a pilot of the field study on an earlier version of the GenderMag Recorder's Assistant that did not list “none of the above” as a checkbox option, participants did sometimes mark facets that they never discussed verbally or in writing.) To look for evidence of “brainful” engagement or lack thereof, we measured whether, for each facet they *checked off*, the videotaped teams gave other evidence of commitment to it via either a mention in their free-form response areas or a verbalization on the videorecordings. This measure showed engagement in 80-87% of the facets they marked.

An alternative lens on depth of engagement lies in what the teams actually said to one another about Abby. Some participants referred to Abby at a very surface level, with no information content about Abby. For example, in the quote below, the information content is not about Abby herself, but rather about the choices available:

Paper2-P2: "I'd say she will know what to do at this step because there's only 3 choices, 'yes', 'no', or 'cancel'...."

In contrast, some participants gave real attention to how Abby worked through the ways her facets led to her choices:

Tool2-P2: "And then I would also say willingness to tinker. Because she's not going to be willing to tinker with the screen to find out if it's the right screen or not."

To get a sense for teams' depth of engagement with Abby, we analyzed the videorecorded teams' verbalizations with explicit content about *both* Abby and her facets in a single conversational turn, like the one above. As Fig. 6 shows, the Tool teams showed much more evidence (via Abby-information content) of

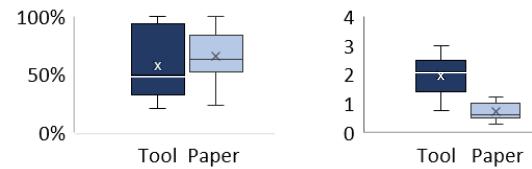


Fig. 5 Engagement: (Left:) Tool vs. Paper teams' mentions of Abby as a % of their written CW responses (no significant difference). (Right:) Number of facets per response: Tool teams mentioned significantly more facets/response.

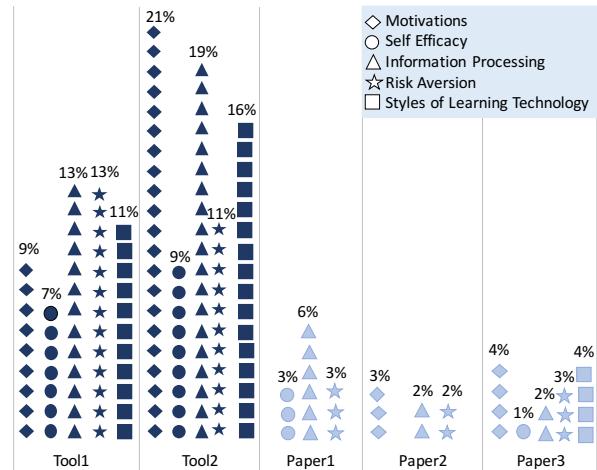


Fig. 6: How often each videorecorded Tool or Paper team verbalized Abby-information content, broken down here by facet, as a percent of their Abby-mentions. The Tool teams showed deeper engagement than the Paper teams.

engagement depth with Abby than the Paper teams did.

Given that the Paper teams' engagement was as strong as the strongest prior work we have been able to find on persona engagement, we expected a “ceiling” effect; i.e., we did not see room for much more engagement. However, the Tool teams surprised us. As Table VII summarizes, Paper teams were strong with engagement, but Tool teams were stronger.

VI. DISCUSSION

Our results show that whether to “tool up” a sociotechnical Theory-of-Mind method like GenderMag is not a simple question. As Fig. 7 summarizes, our results revealed a checkerboard of complementary strengths in Paper vs. Tool.

A. Are the strengths transferable...?

Some of the strengths in supporting our participants may be due in part to the way each was presented (i.e., not inherent to tools or paper), and this suggests that tools or paper could obtain some of the strengths demonstrated by the other. As an example of tool-to-paper transferability, the tool’s checkboxes seemed to remind participants of the facets. This could be implemented in the paper version by adding the same checkboxes to the paper form. An example of paper-to-tool transferability is that paper Abby made all of Abby’s details readily available; this could be accomplished by adding a second display screen to a tool’s setup, so that Abby’s complete details could always be displayed.

B. ...or Inherent?

However, there are some strengths that may be inherent to what tool support vs. paper support can bring to a sociotechnical Theory-of-Mind method. For example, paper as a medium (1) brings less cognitive load, and cognitive load works against Theory-of-Mind [53]. Also, (2) the paper medium is tied to enhanced comprehension of written material [1], which is needed for empathy and engagement with personas like Abby, whose existence is solely in the form of a written description. This may explain why Tool teams so often turned to “paper Abby.”

The Tool condition also brought key advantages to our participants that seem tied to the medium—e.g., the continually updated screen display. Recall that the Tool teams (with access to paper Abby) had greater depth of engagement with Abby than

TABLE VII: SUMMARY OF TOOL VS. PAPER ENGAGEMENT STRENGTHS IN SUPPORTING OUR PARTICIPANTS.

	Abby engagement	Facet engagement	Depth of engagement
Tool strengths	High (Study #1, Study #2)	Tool: more engagement (Study #2)	Tool: greater depth (Study #2)
Paper strengths	High (Study #2)		

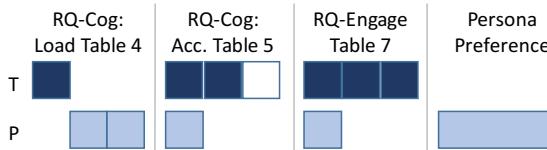


Fig. 7: Visual summary of Tool vs. Paper strengths (summarizes Tables 4, 5, and 7, plus the preference for paper-based Abby).

Paper teams did (also with access to paper Abby). We are again reminded of field study participant West1’s observation that the tool “keeps you focused.” The tool enabled a coordinated display of what exact action in the prototype was being evaluated involving what widget/feedback, and what had been said about it. This may help explain the Tool participants’ rapid recovery from detours (recall Table IV).

C. Social aspects

The social aspects of the tool seemed to help our participants with recording accuracy. GenderMag sessions occur in group settings (e.g., a conference room). In the paper-based method, one team member usually projects the prototype, and the rest of the team discusses the action they see playing out on the projector while the recorder somehow captures the discussion (using paper or word-processing on another computer). However, in the tool-based setting, the prototype step and CW questions are integrated on the projection screen, so the entire team can see what is being recorded for what action at the same time. This transparency may have been another reason for the Tool teams’ better accuracy: if others are watching, a recorder may be more vigilant in capturing what they say, and other team members will have more opportunity to catch recording errors right away.

VII. CONCLUDING REMARKS

The complementary strengths of each medium suggests that the best ways to support developers’ use of a method like GenderMag lie in strategic partnerships of tooling and paper.

The first category of strengths in Fig. 7, cognitive load, seems challenging to resolve because of the interdependencies among how stress (Paper was better), perceived ease of use (Tool was better), and feelings of success (Paper was better) interact with one another and with cognitive absorption/focus, engagement, and Theory-of-Mind processing [45, 52, 53]. How to go about resolving this tension is an open question.

The second and third categories yield more obvious possibilities. Accuracy needed work in both conditions (so no “good” choice here), but one commonality was a single recorder capturing everyone’s ideas in real time. Perhaps distributing the recording to all team members and then sharing/combining what they wrote would improve accuracy on either medium. Engagement, on the other hand, was good in both conditions (no “bad” choice here). Still, the tool was better; perhaps the paper medium’s facet engagement might be further improved by adding facet checkboxes to the paper forms, as mentioned earlier.

The fourth category, Personas, yields a clear choice. The participants’ preferences, their ability to deeply engage with Abby, comprehend written material [1], and to learn and think about her facets [45], all point to paper personas.

Thus, the key is to find the right combinations of tools and paper to best support a sociotechnical Theory-of-Mind method like GenderMag, to enable software teams to create more human-centric, adaptable, and usable software for everyone.

The GenderMag Recorder’s Assistant is an Open Source project, and we welcome contributions. To download it or contribute to it, go to <http://gendermag.org>.

REFERENCES

- [1] R. Ackerman and T. Lauterman. Taking reading comprehension exams on screen or on paper? A metacognitive analysis of learning texts under time pressure. *Computers in Human Behavior* 28(5), pp. 1816-1828, 2012.
- [2] T. Adlin and J. Pruitt, *The Essential Persona Lifecycle: Your Guide to Building and Using Personas*, Morgan Kaufmann/Elsevier, 2010.
- [3] L. Beckwith and M. Burnett, Gender: An important factor in end-user programming environments? *IEEE VL/HCC*, pp. 107-114, 2004.
- [4] L. Beckwith, M. Burnett, S. Wiedenbeck, C. Cook, S. Sorte, and M. Hastings, Effectiveness of end-user debugging software features: Are there gender issues? *ACM CHI*, pp. 869-878, 2005.
- [5] L. Beckwith, C. Kissinger, M. Burnett, S. Wiedenbeck, J. Lawrence, A. Blackwell, and C. Cook, Tinkering and gender in end-user programmers' debugging, *ACM CHI*, pp. 231-240, 2006.
- [6] M. Blackmon, P. Polson, M. Kitajima, and C. Lewis, Cognitive walkthrough for the web, *ACM CHI*, pp. 463-470, 2002.
- [7] M. Blackmon, P. Polson, and C. Lewis, Automated Cognitive Walkthrough for the Web (AutoCWW), *ACM CHI Workshop: Automatically Evaluating the Usability of Web Sites*, 2002.
- [8] M. Burnett, L. Beckwith, S. Wiedenbeck, S. D. Fleming, J. Cao, T. H. Park, V. Grigoreanu, and K. Rector, Gender pluralism in problem-solving software, *Interacting with Computers* 23(5), pp. 450–460, 2011.
- [9] M. Burnett, R. Counts, R. Lawrence, H. Hanson, Gender HCI and Microsoft: Highlights from a longitudinal study, *IEEE VLHCC*, pp. 139-143, 2017.
- [10] M. Burnett, S. D. Fleming, S. Iqbal, G. Venolia, V. Rajaram, U. Farooq, V. Grigoreanu, and M. Czerwinski, Gender differences and programming environments: Across programming populations, *IEEE Symp. Empirical Soft. Eng. and Measurement*, Article 28 (10 pages), 2010.
- [11] M. Burnett, A. Peters, C. Hill, and N. Elarie, Finding gender inclusiveness software issues with GenderMag: A field investigation, *ACM CHI*, pp. 2586-2598, 2016.
- [12] M. Burnett, S. Stumpf, J. Macbeth, S. Makri, L. Beckwith, I. Kwan, A. Peters, and W. Jernigan, GenderMag: A method for evaluating software's gender inclusiveness. *Interacting with Computers* 28(6), pp. 760-787, 2016.
- [13] M. Burnett, S. Stumpf, L. Beckwith, and A. Peters, The GenderMag Kit: How to Use the GenderMag Method to Find Inclusiveness Issues through a Gender Lens, <http://gendermag.org/> 2018.
- [14] P. Cafferata and A. M. Tybout, Gender differences in information processing: a selectivity interpretation, in *Cognitive and Affective Responses to Advertising*, Lexington Books, 1989.
- [15] J. Cao, K. Rector, T. Park, S. Fleming, M. Burnett, and S. Wiedenbeck, A debugging perspective on end-user mashup programming, *IEEE VLHCC*, pp. 149-159, 2010.
- [16] J. Cassell, Genderizing HCI, In *The Hand-book of Human-Computer Interaction*, M.G. Helander, T.K. Landauer, and P.V. Prabhu (eds.). L. Erlbaum Associates Inc., pp. 402-411, 2002.
- [17] S. Chang, V. Kumar, E. Gilbert, and L. Terveen, Specialization, homophily, and gender in a social curation site: findings from Pinterest, *ACM CSCW*, pp. 674-686, 2014.
- [18] G. Charness and U. Gneezy, Strong evidence for gender differences in risk taking, *J. Economic Behavior & Organization* 83(1), pp. 50–58, 2012.
- [19] A. Cooper, *The Inmates Are Running the Asylum*, Sams Publishing, 2004.
- [20] C. Coursaris, S. Swierenga, and E. Watrall, An empirical investigation of color temperature and gender effects on web aesthetics, *J. Usability Studies* 3(3), pp. 103-117, May 2008.
- [21] S. Cunningham, A. Hinze and D. Nichols, Supporting gender-neutral digital library creation: A case study using the GenderMag Toolkit, *Digital Libraries: Knowledge, Information, and Data in an Open Access Society*, pp. 45-50, 2016.
- [22] A. Dingli and J. Mifsud, USEful: A framework to mainstream web site usability thorough automated evaluation, *Int. J. Human Computer Interaction* 2(1), pp. 10-30, 2011.
- [23] T. Dohmen, A. Falk, D. Huffman, U. Sunde, J. Schupp, G. Wagner. Individual risk attitudes: Measurement, determinants, and behavioral consequences, *J. European Econ. Assoc.* 9(3), pp. 522–550, 2011.
- [24] A. Durndell and Z. Haag, Computer self efficacy, computer anxiety, attitudes towards the Internet and reported experience with the Internet, by gender, in an East European sample, *Computers in Human Behavior* 18, pp. 521–535, 2002.
- [25] E. Friess, Personas and decision making in the design process: an ethnographic case study, *ACM CHI*, pp. 1209-1218, 2012.
- [26] V. Grigoreanu and M. Mohanna, Informal cognitive walkthroughs (ICW): paring down and pairing up for an agile world, *ACM CHI*, pp. 3093-3096, 2013.
- [27] J. Hallström, H. Elvstrand, and K. Hellberg, Gender and technology in free play in Swedish early childhood education, *Int. J. Technology and Design Education* 25(2), pp. 137-149, 2015.
- [28] S. Hart, and L. Staveland, Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research, *Advances in Psychology* 52, pp. 139–183, 1988.
- [29] K. Hartzel, How self-efficacy and gender issues affect software adoption and use, *Commun. ACM* 46(9), pp. 167–171, 2003.
- [30] C. Hill, S. Ernst, A. Oleson, A. Horvath and M. Burnett, GenderMag experiences in the field: The whole, the parts, and the workload, *IEEE VL/HCC*, pp. 199-207, 2016.
- [31] C. Hill, M. Haag, A. Oleson, C. Mendez, N. Marsden, A. Sarma, and M. Burnett, Gender-inclusiveness personas vs. stereotyping: Can we have it both ways? *ACM CHI*, pp.6658-6671, 2017.
- [32] W. Hou, M. Kaur, A. Komlodi, W. Lutters, L. Boot, S. Cotten, C. Morrell, A. Ant Ozok, and Z. Tufekci, Girls don't waste time: Pre-adolescent attitudes toward ICT, *ACM CHI*, pp. 875-880, 2006.
- [33] A. Huffman, J. Whetten, and W. Huffman, Using technology in higher education: The influence of gender roles on technology self-efficacy, *Computers in Human Behavior* 29(4), pp. 1779–1786, 2013.
- [34] N. Jacobsen, and B. John, Two case studies in using cognitive walkthrough for interface evaluation (No. CMU-CS-00-132), Carnegie-Mellon Univ School of Computer Science, 2000.
- [35] T. Judge, T. Matthews, and S. Whittaker, Comparing collaboration and individual personas for the design and evaluation of collaboration software, *ACM CHI*, pp. 1997-2000, 2012.
- [36] C. Kelleher, Barriers to programming engagement, *Advances in Gender and Education* 1, pp. 5-10, 2009.
- [37] T. Mahatody, M. Sagar, and C. Kolski, State of the art on the cognitive walkthrough method, its variants and evolutions, *Int. J. Human-Computer Interaction* 26(8), pp. 741-85, 2010.
- [38] J. Margolis and A. Fisher, *Unlocking the Clubhouse: Women in Computing*, MIT Press, 2003.
- [39] N. Marsden and M. Haag, Evaluation of GenderMag personas based on persona attributes and persona gender, *HCI International 2016 – Posters' Extended Abstracts: Proceedings Part I*, pp. 122-127, 2016.
- [40] C. Mendez, H. S. Padala, Z. Steine-Hanson, C. Hilderbrand, A. Horvath, C. Hill, L. Simpson, N. Patil, A. Sarma, M. Burnett, Open Source barriers to entry, revisited: A sociotechnical perspective, *ACM/IEEE ICSE 2018*.
- [41] J. Meyers-Levy, B. Loken, Revisiting gender differences: What we know and what lies ahead, *J. Consumer Psychology* 25(1), pp. 129-149, 2015.
- [42] J. Meyers-Levy, D. Maheswaran, Exploring differences in males' and females' processing strategies, *J. Consumer Research* 18, pp. 63–70, 1991.
- [43] J. Nielsen, Enhancing the explanatory power of usability heuristics. *ACM CHI '94*, pp.152-158, 1994.
- [44] A. O'Leary-Kelly, B. Hardgrave, V. McKinney, and D. Wilson, The influence of professional identification on the retention of women and racial minorities in the IT workforce, *NSF Info. Tech. Workforce & Info. Tech. Res. PI Conf.*, pp. 65-69, 2004.
- [45] S. Oviatt, Human-centered design meets cognitive load theory: Designing interfaces that help people think. In *Proceedings of the 14th ACM International Conference on Multimedia*, pp. 871-880, 2006. <https://doi.org/10.1145/1180639.1180831>

- [46] Piazza Blog, STEM confidence gap. Retrieved September 24th, 2015, <http://blog.piazza.com/stem-confidence-gap/>
- [47] D. Premack and G. Woodruff. Does the chimpanzee have a theory of mind? *Behavior & Brain Sciences* 1(4), pp. 515-526, 1978.
- [48] J. Pruitt and J. Grudin, Personas: practice and theory. ACM DUX, pp. 1-15, 2003.
- [49] R. Riedl, M. Hubert, and P. Kenning, Are there neural gender differences in online trust? An fMRI study on the perceived trustworthiness of eBay offers, MIS Quarterly 34(2), pp. 397-428, 2010.
- [50] J. Rieman, S. Davies, D. Hair, M. Esemplare, P. Polson and C. Lewis, An automated cognitive walkthrough, ACM CHI, 1991.
- [51] D. Rosner and J. Bean, Learning from IKEA hacking: I'm not one to decoupage a tabletop and call it a day, ACM CHI, pp. 419-422, 2009.
- [52] R. Saadé and B. Bahli. The impact of cognitive absorption on perceived usefulness and perceived ease of use in on-line learning: An extension of the Technology Acceptance Model. *Information & Management* 42(2), pp. 317-327, 2005.
- [53] D. Schneider, R. Lam, A. Bayliss, P. Dux. 2012. Cognitive load disrupts implicit theory-of-mind processing. *Psychological Science* 23(8), pp. 842-847, 2012.
- [54] A. Sears, Heuristic walkthroughs: Finding the problems without the noise, Int, J. Human-Computer Interaction 9(3), pp. 213- 234, 1997.
- [55] A. Shekhar and N. Marsden. Cognitive Walkthrough of a learning management system with gendered personas. 4th Gender & IT Conference (GenderIT'18), pp. 191-198, 2018. doi:10.1145/3196839.3196869
- [56] S. Simon, The impact of culture and gender on web sites: An empirical study, The Data Base for Advances in Information Systems 32, pp. 18-37, 2001.
- [57] A. Singh, V. Bhaduria, A. Jain, and A. Gurung, Role of gender, self-efficacy, anxiety and testing formats in learning spreadsheets, *Computers in Human Behavior* 29(3), pp. 739-746, 2013.
- [58] R. Spencer, The streamlined cognitive walkthrough method, working around social constraints encountered in a software development company, ACM CHI, pp. 353-359, 2000.
- [59] E. Weber, A. Blais, and N. Betz, A domain-specific risk-attitude scale: Measuring risk perceptions and risk behaviors, *J. Behavioral and Decision Making* 15, pp. 263-290, 2002.
- [60] C. Wharton, J. Rieman, C. Lewis, and P. Polson, The cognitive walkthrough method: A practitioner's guide. In *Usability Inspection Methods*, pp. 105-140, 1994.

Searching Over Search Trees for Human-AI Collaboration in Exploratory Problem Solving: A Case Study in Algebra

Benjamin T. Jones and Steven L. Tanimoto

Paul G. Allen School of Computer Sci. and Engr.

University of Washington

Seattle, Washington 98195

Email: benjones@cs.washington.edu and tanimoto@cs.washington.edu

Abstract—Artificial intelligence and machine learning work very well for solving problems in domains where the optimal solution can be characterized precisely or in terms of adequate training data. However, when humans perform problem solving, they do not necessarily know how to characterize an optimal solution. We propose a framework for human-AI collaboration that gives humans ultimate control of the results of a problem solving task while playing to the strengths of the AI by persisting an agent’s search trees and allowing humans to explore and search this search tree. This allows the use of AI in exploratory problem solving contexts. We demonstrate this framework applied to algebraic problem solving, and show that it enables a unique mode of interaction with symbolic computer algebra through the automatic completion and correction of traditional derivations, both in digital ink and textual keyboard input.

I. INTRODUCTION

Computer algebra systems are a common form of artificial intelligence (AI) problem solving. They excel at well-defined problems such as solving an equation for a variable, minimizing a function, or simplifying complex expressions, but do not have affordances for more exploratory work. This limits their usefulness for mathematicians, physicists, and engineers, who do much of their mathematical work in an ideation phase in which there is not a distinct goal. In this phase of their work, these professionals stick to whiteboards and paper, complaining that computer algebra systems inhibit their creativity by over-constraining input [1]. Professionals also cite a lack of transparency about the steps taken by the program, and the lack of 2D (traditional handwritten notation) input when asked why they avoid these systems. Several attempts have been made to solve the input problems [2] [3] [4], but none of these systems have seen any professional-level adoption, not having the power of a more general CAS [5].

The problem of designing a computer algebra system for exploratory work is an instance of the more general problem of designing an interface for an AI problem solving tool for contexts in which there is no known or easily expressible goal. Traditional AI search requires a goal state or optimization function for direction, but in exploratory problem solving we envision a more interactive system in which the human and AI act as collaborators: the human providing intuition and constantly refining the search direction, while the AI leverages

its speed and memory to quickly and thoroughly explore large parts of the search space.

As a step towards this vision, we set out to design and build a prototype for an exploration-focused computer algebra system that would address the problems identified by professional users. From this design exercise, this work makes three contributions:

- a new interaction technique for CAS that leverages the full power of a professional CAS while allowing maximal freedom of expression,
- insights on how to interpret user intent in such a collaborative system, and
- a framework for designing human-AI collaborative systems that extends to contexts beyond algebra.

II. PRIOR WORK

Prior work has explored human-human collaboration in problem solving, with computational support. CoSolve allowed humans to collaboratively explore a problem space [6]. In doing so, they generated a search tree much as a traditional AI would, but unlike in an AI system, this tree was persistent and exposed to all collaborators so that they could look at branches others had explored to gain an understanding of the entire space of possible partial and candidate solutions.

In a human-plus-agent collaboration context, Lalanne and Pu showed how a mixed-initiative system could support interactive formulation and solution of constraint satisfaction problems [7]. Their visualizations allowed a user to get a sense not only of how constrained the problem was, as formulated, but of the dynamics of the algorithm searching for solutions.

Another form of human-plus-agent collaboration involves the joint exploration of a search tree or graph. For example, an agent could perform much the same task as humans exploring the state space in CoSolve. An AI agent can do this much faster than a human. Furthermore, the trees generated in this manner can easily be far too large for a human to manually explore. In order to enable exploration in this very large solution space, we suggest adopting methods from search in the information-retrieval sense.

Conceptually, this moves from a problem solving environment in which people take many small steps one at a time, to one in which an AI system takes many small steps one at a time, and a human takes a few very large steps in order

to explore the space and eventually converge on a desired solution.

III. UNDERLYING PRINCIPLES

Here we briefly describe the theoretical foundation for our work, and introduce terminology that we use later to explain how our system functions.

A. Classical Theory

By the “classical theory of problem solving” we mean the conceptualization of problem solving as state-space search and the associated terminology and techniques that support its actualization. Nilsson’s presentation is a particularly well-scorched one that we can point to [8].

A “state” of a problem is an encapsulation of the information essential to the problem solving process at a snapshot in time of a solving process. In the context of algebraic problem solving, the state is a collection of one or more algebraic expressions that define the relationships among relevant variables and constants.

A transformation is a potential action that might be applied to a state to produce a new state. In our context, transformations are the rules of algebra (e.g. “multiply both sides of an equation by a value”), expressed as a collection of rewrite rules. Algebra is an example of a problem domain with an infinite space of operators; for example, the value multiplying both sides could be any constant.

A goal state is one that satisfies all the criteria for a solution to the problem or that is simply the required end state for a sequence of transformations that might be considered to be the real solution. For exploratory computer algebra, the true criteria for the goal state are unknown; the overall goals are to gain an understanding of the system being explored and to find useful representations of that system. Thus the criteria for a solution will be constantly evolving for the user.

In the classical theory, the process of solving a problem is a search process. It usually starts with some representation of the initial state, applies transformations repeatedly to get new states, and tests the new states to find out if a goal state has been reached. However, some search techniques work differently, possibly searching backwards from a goal state or making jumps around the state space using prior knowledge.

B. Problem-space Graph

An essential abstraction in the classical theory is the notion of “problem-space graph.” This parts of this graph are implied by the parts of a well-formed problem, as formulated in terms of the theory. There is a graph vertex for each possible state of the problem, and there is an edge (v_i, v_j) whenever one of the transformations of the problem maps v_i to v_j . This graph is usually infinite, and “platonic” in the sense that it is implicit and not represented explicitly except in relatively small parts as described below.

C. Explicit Subgraphs

Given a state s and a maximum depth d , a subset $S_{s,d}$ of states can be defined as all states reachable from s by applying sequences of operators of lengths less than or equal to d . Taking the vertices corresponding to $S_{s,d}$ of the problem-space graph, together with the edges that interconnect them gives us the explicit subgraph for s and d . In our implementation, an AI system will generate explicit subgraphs on demand for a succession of current states s .

An example of an explicit subgraph for an algebraic problem-solving context is shown in Fig. 1.

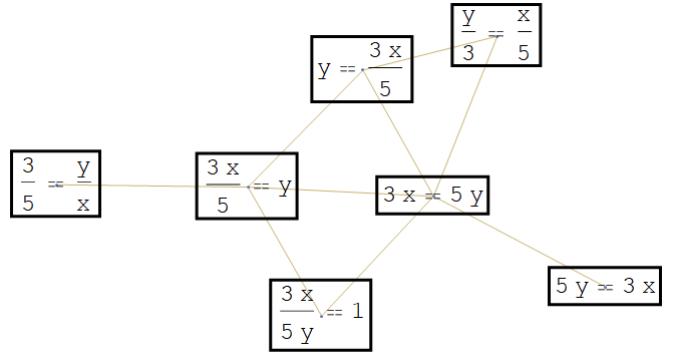


Fig. 1. An explicit subgraph for algebra problem solving, where the current state corresponds to the formula $\frac{3x}{5} = y$.

D. Querying the Graph

The explicit subgraph, in our system, represents a rich neighborhood of the current state that offers possible new states to the user. However, this graph is itself typically very large – large enough that it cannot be fully displayed to the user. We offer the user an opportunity to interact with the explicit subgraph through a query-and-retrieval process. Users have great freedom when entering a query. It is typically performed with digital ink (described later) and may represent either a complete or an incomplete algebraic expression.

E. Retrieval Process

Finally, there is another kind of search process that involves comparing the query with vertices in the explicit subgraph, in order to return the most relevant formulas, given the user’s query and current state. While the process is itself a form of search, we will refer to this as querying later, to distinguish it from other types of search.

IV. CASE STUDY

Our prototype has been directed towards the goal of making it easy for users to perform exploratory mathematics activities with computer support. In designing our system, therefore, we wish to give users access to the underlying computer algebra system while allowing maximum expressiveness in the user’s input. To do this, we allow input in the form of digital ink.

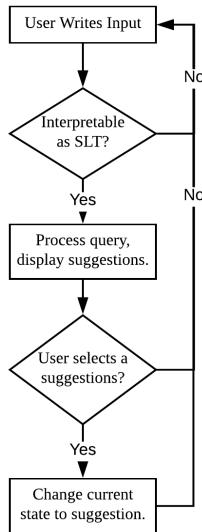


Fig. 2. The user interaction model. This diagram assumes that there is already a current state from which the CAS is exploring.

A. Digital Ink for Computer Algebra Systems

The vast majority of input possible with ink is not directly interpretable as expression trees, the preferred data format for CASs. Mathematics handwriting recognition can help bridge this gap. Most systems do not produce CAS compatible expressions as output, however, but rather trees representing the structural layout of symbols (numbers, operators, variables, etc.) on the page. We follow Stalnaker and Zanibbi in calling these symbol layout trees [9]. An example of the symbol layout tree for the expression $\frac{3x}{5} = y$ is shown in Fig. 3. Even with these trees, there are still many inputs that cannot be interpreted directly as CAS expressions such as $x =$, $\underline{3x} = y$, or instances of mismatched parenthesis. All of these are examples of things we would expect somebody to write when they are exploring, either when in the process of writing or by mistake.

The existence of these common inputs that current computer algebra systems cannot handle suggests a design niche for our system - interfacing with a computer algebra system via incomplete or incorrect inputs. The system will interpret inputs from the user as attempts to write a correct expression (where correctness is determined by equivalence an earlier well formed input used as a starting state). These will be input as queries on the CAS's explored graph, and the results will be presented as completions or corrections for the current line of derivation. Fig. 2 illustrates this interaction flow. Notice that at any stage the user has the ability to continue writing, just as they would on physical media. By only offering suggestions when possible, our system maintains all of the flexibility and expressibility of traditional paper or board work. Fig. 4 shows our prototype user interface, using MyScript handwriting recognition [10] for input and [11] as the CAS.

B. Retrieval of Mathematical Formulas

Mathematics information retrieval (MIR) is a subfield of information retrieval concerned with how to properly index

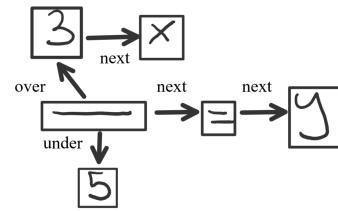


Fig. 3. The symbol layout tree for $\frac{3x}{5} = y$. The symbol-pair tuple of the fraction bar and x is $(\underline{\hspace{1cm}}, x, over, next)$.

and query mathematical content, in opposition to traditional string based content. One insight from this field is that using purely text-based representations of structured symbolic content provides bad querying results because two symbols which are close in a 2-dimensional layout can be far apart in a 1-dimensional string encoding [12]. We will use as our query format the symbol layout trees (SLT) produced by commercial mathematics handwriting recognition software [10], and will borrow our indexing scheme from Tangent, one of the most successful systems in the MIR literature [9]. In this scheme, an inverted index is formed over pairs of symbols, annotated by the path connecting them, called symbol-pair tuples. In our system we drop the edge labels for the indexing step to improve recall. Our ranking function also derives from Tangent's, but with a significant modification in order to account for the context in which the query is made. In user studies Stalnaker and Zanibbi found that a standard f-measure of precision and recall over pairs produced the best ranking results [9]. The initial version of our system used this measure as well, but we found that the ranking results were counterintuitive. We eventually realized that the mismatch between expectation and results was due to portions of an expression that remain unchanged between lines of a derivation dominating the similarity score. What a user expects is that the *changes* they have made to an expression are the most important parts of the query.

To illustrate this with a simple example, consider the expression $\frac{3x}{5} = y$. A user is attempting to move the 5 to the other side of the equation, to obtain $3x = 5y$. The first thing the user does is erase the 5. This results in the query $\frac{3x}{y} =$. Using a simple f-measure ranks $\frac{3x}{y} = 5$ above $3x = 5y$ due to the shared structure around the fraction bar, despite the fact that this result ignores the only explicit change that the user made.

In order to take into account the context in which the query is being made, we again use a simple f-measure, biased towards recall, but rather than looking at the symbol pairs of the query and a proposed result, we consider the pair edits between the last state of the derivation and the query or result. A pair edit is either the addition, or removal of a pair. In our example, the query becomes a singleton set of symbol-pair tuples: $\{(\underline{\hspace{1cm}}, 5, under, removed)\}$. Since the erroneous result we previously received does not remove 5 from this location, it scores 0 for recall and is therefore ranked lower than the result we expected.

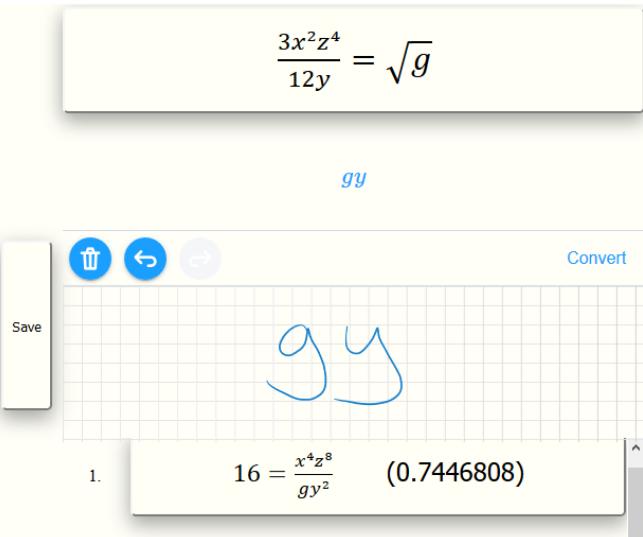


Fig. 4. Our prototype interface. From top to bottom: the current state of the derivation, current handwriting recognition result, canvas for handwritten input, results list. The “save” button sends the current recognition results to the CAS to start a new derivation. Here the user has indicated that *gy* should appear together in the results. The top result has a high similarity score: 0.744.

V. DISCUSSION

Most computer algebra systems work by interpreting string input as particular commands and applying them (either as a single transformation such as “factor”, or “fourier transform”, or as the specification of an optimization for an AI search as in “simplify” or “solve”). In either case, the string input must be unambiguous making these systems too rigid for exploratory work. Traditional mathematical notation does not work like this; ambiguity is normal and humans can resolve it using contextual cues.

Our system flips this order on its head. Rather than interpreting input (matching inputs to exact commands), it explores a search space and generates many inputs that could match the same expression in the state space. This allows for multiple notational schemes to coexist and for the system to switch among them without user intervention, freeing the user to use whichever convention seems most convenient. New notations can even be added to the system on-the-fly by writing new pretty-printing functions. This is much easier than typical UI programming and also easier than designing an input language, since one does not need to worry about ambiguity with existing notation. The meaning is inferred from context (what was the originally inputted expression), just as human mathematicians use context to deal with notational ambiguity.

The overall effect of this switch is that the process of algebraic derivation stops being one of local path planning (choosing the next transformation to try one at a time) and instead becomes one of global planning; the user starts writing expressions that have useful forms or properties, and the CAS will search for true (equivalent) expressions that match these queries. The AI takes on the error-prone, time-consuming, and tedious task of applying transformations and exploring

branches of the search tree to allow humans to take giant leaps through state space rather than small steps.

This key idea can be applied to problem-solving domains other than algebra. Abstracting out the main pieces of the design, we arrive at a framework with three components: an I/O layer, an underlying state-space solver, and an intermediate representation. In our system, these components are math handwriting recognition, CAS, and symbol layout trees. Obvious generalizations include other symbolic reasoning systems such as for chemical formulas and diagrammatic languages, but any problem that can be fit into the classical model will work. In a game-playing context, the IO layer could be a board with pieces at arbitrary coordinates, the solver a chess simulator, and the intermediate representation chess-board notation. Such a system could be used as a chess training tool; a human would start with a known game state, and moves pieces to a position of interest, thinking at a high level. For example, One wants to corner the opponent using specific pieces. The solver would then find a realizable path to a similar board state that implements the player’s general strategy.

Another powerful idea enabled by this decomposition is interchangeability in these layers. In algebra, we can swap a LaTeX frontend for handwriting recognition, allowing inline correction of mathematical documents as they are being written. Entirely non-symbolic representations of equations could also be used, e.g., graphs; recent work in graph querying has shown how to match imperfect hand-drawn graphs to data series by extracting certain features [13].

Implementing a system using this framework requires relatively little coding, and it is extensible by non-experts. In our algebraic context, new notation can be implemented by any end user capable of writing a function that generates intermediate representation from CAS expressions; this is just half of a pretty printer! This ease of implementation extends to other contexts since data display is inherently simpler than data input and interpretation (there is no need to deal with ambiguity in the output direction). In our system, expressing an abstract syntax tree as LaTeX is sufficient. The other required function translates the user’s input into the intermediate representation. One can often rely on existing libraries and interfaces for this functionality (handwriting recognition, text editors, etc.), simplifying the job.

VI. CONCLUSION

In this paper we have proposed a method of human-AI collaborative problem solving via query-based exploration of an AI search tree. We demonstrated this approach through a prototype for algebraic problem solving, and in designing this process ascertained two key design considerations: an intermediate representation between user input and an AI system’s underlying representation should be chosen to express queries in order to maximize expressiveness while keeping the search problem tractable, and that the context of the user’s exploration is crucial to the ranking of results. Finally, we observe that this design pattern is easy to implement and should generalize well across many problem domains.

REFERENCES

- [1] A. Bunt, M. Terry, and E. Lank, “Friend or Foe?: Examining CAS Use in Mathematics Research,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI ’09, (New York, NY, USA), pp. 229–238, ACM, 2009.
- [2] G. Labahn, E. Lank, S. MacLean, M. Marzouk, and D. Tausky, “Math-brush: A system for doing math on pen-based devices,” in *Document Analysis Systems, 2008. DAS’08. The Eighth IAPR International Workshop on*, pp. 599–606, IEEE, 2008.
- [3] R. Zeleznik, T. Miller, C. Li, and J. J. LaViola, “MathPaper: Mathematical Sketching with Fluid Support for Interactive Computation,” in *Smart Graphics*, Lecture Notes in Computer Science, pp. 20–32, Springer, Berlin, Heidelberg, Aug. 2008.
- [4] R. Zeleznik, A. Bragdon, F. Adeputra, and H.-S. Ko, “Hands-on Math: A Page-based Multi-touch and Pen Desktop for Technical Work and Problem Solving,” in *Proceedings of the 23rd Annual ACM Symposium on User Interface Software and Technology*, UIST ’10, (New York, NY, USA), pp. 17–26, ACM, 2010.
- [5] A. Bunt, M. Terry, and E. Lank, “Challenges and Opportunities for Mathematics Software in Expert Problem Solving,” *Human-Computer Interaction*, vol. 28, pp. 222–264, May 2013.
- [6] S. B. Fan, T. Robison, and S. L. Tanimoto, “CoSolve: A system for engaging users in computer-supported collaborative problem solving,” in *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 205–212, Sept. 2012.
- [7] P. Pu and D. Lalanne, “Design Visual Thinking Tools for Mixed Initiative Systems,” in *Proceedings of the 7th International Conference on Intelligent User Interfaces*, IUI ’02, (New York, NY, USA), pp. 119–126, ACM, 2002.
- [8] N. J. Nilsson, *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill Pub. Co., 1971.
- [9] D. Stalnaker and R. Zanibbi, “Math expression retrieval using an inverted index over symbol pairs,” in *Document recognition and retrieval XXII*, vol. 9402, p. 940207, International Society for Optics and Photonics, 2015.
- [10] S. MyScript, *MyScript Cloud Development Kit*.
- [11] W. R. Inc, *Mathematica, Version 11.2*.
- [12] S. Kamali and F. W. Tompa, “Structural Similarity Search for Mathematics Retrieval,” in *Intelligent Computer Mathematics*, Lecture Notes in Computer Science, pp. 246–262, Springer, Berlin, Heidelberg, July 2013.
- [13] M. Mannino and A. Abouzied, “Expressive Time Series Querying with Hand-Drawn Scale-Free Sketches,” in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI ’18, (New York, NY, USA), pp. 388:1–388:13, ACM, 2018.

Expresso: Building Responsive Interfaces with Keyframes

Rebecca Krosnick¹, Sang Won Lee¹, Walter S. Lasecki^{1,2}, Steve Oney^{2,1}

¹Computer Science & Engineering, ²School of Information

University of Michigan | Ann Arbor, MI USA

{rkros,snaglee,wlasecki,soney}@umich.edu

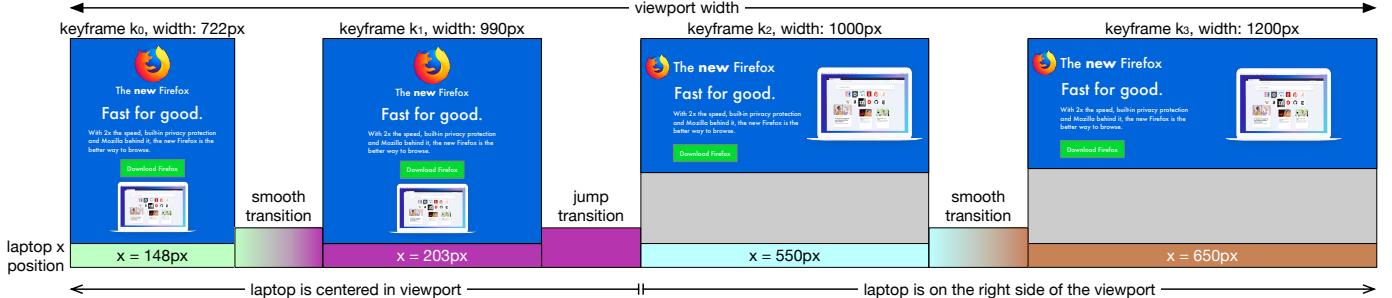


Fig. 1: Espresso allows users to create responsive interfaces by defining keyframes (which specify how the interface should look for a particular viewport size) and specifying how element properties should transition between keyframes. In this illustration, there are four keyframes (from left to right: k_0 , k_1 , k_2 , and k_3). k_0 and k_1 specify that the page layout should be stacked vertically and centered for narrow viewports, such as mobile phones. k_2 and k_3 specify a two-column layout for wider viewports, such as full desktop browsers. This illustration highlights the x-position for the laptop image for all four keyframes and how it transitions between keyframes. In the resulting UI, the laptop is centered for viewport widths < 1000 pixels and is on the right side of the page for widths ≥ 1000 pixels.

Abstract—Web developers use responsive web design to create user interfaces that can adapt to many form factors. To define responsive pages, developers must use Cascading Style Sheets (CSS) or libraries and tools built on top of it. CSS provides high customizability, but requires significant experience. As a result, non-programmers and novice programmers generally lack a means of easily building custom responsive web pages. In this paper, we present a new approach that allows users to create custom responsive user interfaces without writing program code. We demonstrate the feasibility and effectiveness of the approach through a new system we built, named Espresso. With Espresso, users define “keyframes”—examples of how their UI should look for particular viewport sizes—by simply directly manipulating elements in a WYSIWYG editor. Espresso uses these keyframes to infer rules about the responsive behavior of elements, and automatically renders the appropriate CSS for a given viewport size. To allow users to create the desired appearance of their page at all viewport sizes, Espresso lets users define either a “smooth” or “jump” transition between adjacent keyframes. We conduct a user study and show that participants are able to effectively use Espresso to build realistic responsive interfaces.

Index Terms—responsive layouts, web programming, CSS

I. INTRODUCTION

Web User Interfaces (UIs) often need to work across a variety of form factors and viewport sizes: from small mobile devices to large high-resolution displays. Web developers use *responsive design* to build websites that can adapt to any viewport size and window configuration. Cascading Style Sheets (CSS)—a language for specifying web

pages’ appearance—supports responsive design through “media queries” (@media), which specify style rules that apply for particular form factors.

CSS is an expressive language but is complex to use, especially in the context of creating responsive designs. This is because: 1) various rules need to be applied to each of multiple elements in the Document Object Model (DOM) hierarchy to achieve a particular visual appearance, 2) developers must be able to envision how new rules interact with existing rules, including from third party libraries, and elements in the context of the given HyperText Markup Language (HTML) hierarchy, and 3) developers must understand how rules affect page appearance across different page sizes and states [1], [2].

In this paper, we present Espresso, a tool that allows users to create custom responsive UIs. Espresso introduces the idea of using *keyframes* to define responsive layouts. Keyframes have had a long and successful history in computer-aided animation [3], where they greatly reduce animators’ workload by allowing a computer to generate smooth transitions between drawings. With Espresso, users define keyframes that specify how a UI should look for a particular viewport size. Espresso then generates a responsive UI that satisfies the layout of every keyframe and infers the layouts for viewport sizes between keyframes. Espresso also gives users control over how their UI should transition between keyframes. In this paper, we contribute the following:

- The idea of defining responsive UI behavior by specifying the UI appearance at specific viewport sizes (keyframes), interpolating (smooth) transitions between them, and sup-

porting discontinuous (jump) transitions between significantly different layout states.

- An instantiation of our idea in Espresso, a system that allows users to encode requirements for a responsive web UI through direct manipulation.
- Evidence from a user study that participants without relevant programming background were able to effectively specify responsive web UIs with Espresso.

II. RELATED WORK

A. Terminology

In our discussion of related work, we differentiate between three types of UI layouts. A *fluid* UI is one where elements' dimensions are proportional to the dimensions of the viewport. An *adaptive* UI is one where the programmer creates a different layout per form factor (e.g., as different HTML files), and the platform determines which layout to serve (e.g., the mobile layout or the tablet layout). A *responsive* UI — the focus of this paper — is one where the programmer creates one UI (e.g., as one HTML file) and specifies rules for how its layout should respond to different viewport sizes. In responsive UIs, individual elements' visibility, size, and position often will change for different viewport sizes. Adaptive UIs generally support less fine-grained control through all viewport sizes.

B. Constraint-Based Layouts

Constraints have long been used in UI specification [4]. Constraints allow developers to specify relationships between elements' visual properties that are maintained automatically. Many UI builders, including XCode [5] and Android Studio [6], allow developers to specify a limited set of layout constraints. Specifically, they allow users to define constraints through visual constraint metaphors, like “springs and struts” that expand and contract with the viewport. These models allow developers to define *fluid* layouts, where elements reside based on the viewport size. However, this model is not appropriate for *responsive* layouts because the constraints that they enable are not expressive enough to support rearranging, moving, or toggling element visibility for different viewport sizes. Although previous research has proposed constraints that could vary by UI and viewport state [7], [8], it is still difficult to author these constraints for responsive UIs. Espresso instead infers constraints for responsive UIs from keyframes.

C. WYSIWYG Interface Builders

“What You See Is What You Get” (WYSIWYG) interface builders allow users to specify a UI layout visually. Interface builders were first proposed in academic research [4], [9] and have achieved widespread commercial usage. Many modern web UI programming tools integrate interface builders including Dreamweaver [10], Webflow [11], and Bootstrap Studio [12]. Each of these tools provide live, editable previews of websites as developers write HTML and CSS. They also provide widgets to view and edit CSS properties. However, none of these tools allow responsive behaviors to be specified through direct manipulation. Although they lower the floor for

developers, creating responsive UIs in these tools still requires conceptual CSS knowledge, as they still use the underlying mechanisms of CSS properties and media queries.

D. Programming by Example Interface Builders

Programming by Example (PbE) (sometimes also known as Programming by Demonstration, or PbD) is a paradigm that allows non-programmers to write programs by giving examples of its behaviors. PbE has been used for a variety of applications, such as dynamic user interface creation [13]–[15], script and function creation [16]–[19], and word processing [20]–[22]. Existing systems have used a variety of user interaction and inferencing techniques [23], [24]. Important aspects of user interaction include how the user creates and modifies a demonstration, how the PbE system provides feedback to the user, and how the user invokes a program [24]. PbE systems also vary in the inferencing techniques they use; some use minimal inferencing (requiring the user to explicitly specify generalizations), while others use simple rule-based inferencing, and even others use Artificial Intelligence (AI) [23]. Espresso uses PbE to create UIs that are responsive to viewport width. End-users create a demonstration, or “keyframe”, for each viewport width they want to explicitly specify the appearance of and then directly manipulate UI elements in the WYSIWYG editor. End-users have the ability to see previously created keyframes and modify them, and they can view the page appearance at different viewport sizes by dragging to resize the viewport. As its inferencing algorithm, Espresso uses linear interpolation of two bounding keyframes to determine page appearance for an intermediate viewport width, adjusting which linear rule is used for a viewport width range when a “jump” transition is specified; “smooth” and “jump” transitions are discussed in detail in the “Transition Behaviors” section below.

The prior PbE work that is most relevant to Espresso are tools that can infer linear constraints between elements and viewports from multiple snapshots or demonstrations: Peridot [13], Inference Bear [25], Chimera [14], and Monet [26]. Although these systems support building fluid UIs, they do not support building responsive UIs, as there is no support for discontinuous jumps between different responsive behavior ranges. Espresso enables building responsive UIs through its smooth and jump transition menu (Fig. 2c).

E. End-User Programming for the Web

More generally, much research has explored end-user programming for the web, including for both custom UI creation and automation, and using a variety of interaction techniques. Chickenfoot [27] allows users to customize existing websites using a simplified language based on UI-oriented keywords and pattern matching. Systems like Inky [28] and CoScripter [29], [30] move closer to supporting natural language interaction with the web, leveraging sloppy programming to allow users to complete and automate web tasks. More recent systems powered by crowdsourcing truly allow end-users to create and interact with web UIs without programming experience. Arboretum [31] allows users to complete web tasks by making

natural language requests and handing off controlled parts of a page to crowd workers for completion. Apparition [32] and SketchExpress [33] enable a user to prototype UI appearance and behavior via natural language and sketch descriptions; this is made possible by crowd workers who fulfill these specifications using WYSIWYG and demonstration tools.

F. Automatic UI Layout

Finally, previous research has proposed automatically generating UI layouts based on developer-specified heuristics [34], [35]. Unlike fully automated UI generation tools, Expresso only generates the transitions between user-specified keyframes, which gives the user more control over the appearance of their interface for different viewport sizes.

III. MOTIVATIONAL CSS STUDY

A. Setup

To better understand the challenges of creating responsive websites, we conducted a study with 8 participants (3 female and 5 male, $\mu = 25$ years old, $\sigma = 5.07$ years). Almost all of our participants were experienced programmers: three participants had at least five years of programming experience in any language, four participants had 2–5 years, and one participant had 6–12 months. Participants had widely varying levels of experience with CSS: two participants had 1–2 years of CSS experience, one had 1–3 months, one had 1–7 days, two had less than one day, and two had no prior CSS experience.

Every participant was given two tasks in an order that was counterbalanced between participants. We adapted our tasks from real-world web pages to ensure that they were realistic. One task involved replicating features of the Mozilla web page (shown in Fig. 1). The other task involved replicating features of a shoe shopping web page (shown in Fig. 2). Both tasks are described in further detail in the Expresso user study task descriptions below, as both tasks were re-used in our evaluation of Expresso.

For each task, participants were given a static (non-responsive) version of the web page and were asked to write CSS to make it responsive. We gave participants animated GIFs demonstrating how the UI should respond to varying width as a user resizes the window. Participants were encouraged to use any online resources (e.g., search engines, tutorials, or libraries) they found helpful and used their preferred code editor. We scheduled study sessions for approximately 1 hour each, and gave participants up to 22 minutes per task to allow time for setup, instructions, and survey.

B. Results and Discussion

We evaluated participants' final web pages according to a rubric (discussed further in the Expresso Evaluation section). Participants achieved a mean accuracy of 45.7% ($\sigma = 23.5\%$). If we calculate task accuracy by participant experience with CSS, we find that the five participants with one week or less CSS experience achieved a mean accuracy of 35.5% ($\sigma = 22.2\%$). The three participants with one month or more of CSS performed better, achieving a mean accuracy of

62.7% ($\sigma = 13.8\%$). These results suggest that, even with the abundant resources (e.g., example code, tutorials, Stack Overflow answers) that are available online, programming responsive UIs is difficult.

To better understand the challenges of writing CSS that participants faced, we analyzed the screen recordings of each participant. One challenge is that a lack of background knowledge makes it difficult to describe the desired rule for a search query. If a participant did not know the name of a relevant CSS keyword, they needed to semantically describe the behavior, which did not always enable them to find the right syntax quickly. For example, in one case, participants were asked to make an element “jump” to the bottom of the page for small page widths. Participants used a variety of search queries, including: “HTML resize to fit screen”, “HTML overflow elements”, “move item to next line responsive”, and “CSS wrap on resize”. These search queries are far from the correct CSS keywords — `flex`, `@media`, or `float`. Further, search terms such as “overflow” may semantically make sense but conflict with an existing CSS property name (i.e., `overflow`). Desired behavior can be easily demonstrated visually (e.g., through a GIF or sketch), but variation in the programmers’ language descriptions and not knowing relevant domain-specific terms can be barriers in searching for answers to responsive web design questions. The challenge of finding the correct CSS keywords and applying them appropriately is exacerbated by the fact that a relatively small set of CSS properties can have widely different effects on a UI’s layout depending on how they are used or combined.

Another challenge was that changing the page’s CSS for one viewport size could affect the layout of other viewport sizes. As a result, the intermediate process of correcting the layout for one viewport size could break the layout for other viewport sizes. The fact that existing code runs the risk of breaking something seemed to be discouraging to participants. During the study, we witnessed many incidents where participants found the right CSS properties to set, but in the end decided not to use them as their initial attempt made the website look worse than it had previously for other viewport sizes.

In sum, this study simulates practical situations where non-professional programmers create an initial version of their website without considering responsive design. The results suggest that even for experienced programmers, it can be challenging to build responsive web pages using CSS.

IV. THE EXPRESSO SYSTEM

We created Expresso to enable people with little to no CSS experience to quickly and easily create responsive web pages. Previous research has found that it is often easier to specify the *appearance* of a web page (how elements should display on a page) than it is to define its *behavior* (how the appearance changes depending on user input and page state) [36]. We designed Expresso to let users define a UI’s responsive behavior by specifying its appearance in a series of keyframes and specifying how the UI should appear in the states between these keyframes. Given this information, Expresso infers how the UI



Fig. 2: The Espresso user interface includes (a) a responsive web page viewing area, (b) a menu for switching between existing keyframes (indexed by their viewport width and height) or switching to a resizable preview mode, and (c) a menu for setting element property values and transition behaviors. Here, the keyframe with viewport width 780 pixels is shown with the pink shoe image element selected. The right menu indicates the current transition behaviors between this keyframe and adjacent ones, namely, a “jump” transition between this keyframe and the next smallest, and a “smooth” transition between this keyframe and the next largest.

should appear for those viewport sizes not explicitly defined by a keyframe. In Espresso, users specify a UI’s appearance for a given keyframe by simply dragging and resizing elements on a visual canvas, similar to how they manipulate objects in drawing or presentation software. This natural interaction allows users to specify complex rules without ever writing display rules or formulae.

A. Adding Keyframes to Make a Website Responsive

The input to Espresso is a static web page, which is represented as one keyframe in Espresso’s user interface. This simulates the scenario where a user wants to modify a static, non-responsive web page to make it responsive. With only one keyframe, the appearance of the web page’s elements in this keyframe applies to all viewport sizes, as this is the only knowledge Espresso has about the web page. When the user adds another keyframe, Espresso’s default behavior is to create a smooth transition gradient between the two keyframes, meaning that elements move and resize linearly between the keyframes. However, the user can customize how their interface transitions between these keyframes, explained more in the Transition Behaviors section. As the user adds more keyframes, Espresso generates a set of piecewise functions representing element property behaviors and the corresponding responsive UI. The user can view the responsive UI by resizing the web page viewport area.

B. Espresso User Interface

The Espresso interface (Fig. 2) consists of a container on the left for viewing the in-progress web page at different viewport widths, a menu at the bottom for navigating existing keyframes and creating new ones, and a menu on the right for modifying element property values and transition behaviors. The user can change the viewport size in which the web page is viewed by selecting a previously created keyframe from the bottom menu or by resizing the viewport via a drag handle. The web page view area is a WYSIWYG editor which allows direct manipulation of elements. When the user has created a new keyframe or selected an existing one, they can then select elements on the page and drag to reposition and

resize them. When an element is selected, the right side menu populates with the element’s properties (e.g., dimensions, position, color), current values, and transition behaviors.

The widgets in the right menu (Fig. 2c) support setting range behaviors through the analogy of colors and gradients. The keyframe currently in view is represented as the turquoise “Current” label, the next smallest keyframe is represented as the magenta “Left” label, and the next largest keyframe is represented as the orange “Right” label. The range between colored labels can be either their color gradient or one solid color as chosen from a dropdown widget. In Fig. 2, there is smooth, linear interpolation behavior between the “Current” keyframe and the “Right” keyframe as represented via the turquoise-to-orange gradient. There is a discontinuity in behavior between the “Left” and “Current” keyframes as represented by a solid color; specifically, the solid color of magenta represents behavior continued from the left range of the “Left” keyframe. Fig. 3 illustrates the element property behaviors that each solid and gradient color option encode. Below, we discuss how to set these transitions and their meaning.

C. Viewport Sizes Between Keyframes

User-created keyframes specify the required UI layout at particular viewport sizes. Espresso infers layouts for the other viewport sizes by inferring how every element property transitions between keyframes. For example, for the “Fast for good” text in Fig. 1, the behaviors for the text’s font size, x-position, and y-position are inferred individually. Together, these inferred property values define the behavior of the text element across different viewport sizes, and the inferred behaviors of all elements together define the page layouts.

D. Transition Behaviors

We infer element property behavior over the range between two adjacent keyframes. By default, we infer a linear interpolation behavior between two adjacent keyframes. For example, in Fig. 1, the laptop has an x-position of $x_2 = 550$ pixels in keyframe k_2 of viewport width $w_2 = 1000$ pixels, and an x-position of $x_3 = 650$ pixels in keyframe k_3 of viewport width $w_3 = 1200$ pixels. Espresso infers a linear interpolation rule

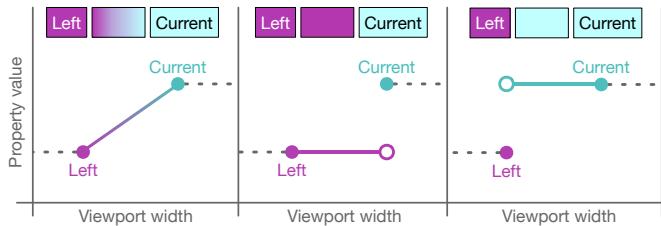


Fig. 3: Graphs illustrating the property behaviors the gradient and solid color dropdown options shown in Fig. 2 encode. Each solid dot represents a keyframe and the line in each graph corresponds to the behavior for the range between the “Left” and “Current” keyframes.

($x = mw + b$) for the laptop x-position for viewport widths $w \in [w_2, w_3]$. The slope m and constant b are calculated based on the (w_2, x_2) and (w_3, x_3) data points provided. Espresso currently only infers linear rules, but other rules, such as higher-order polynomial rules, could be applied under this approach, as we discuss in the Scope section below.

Expresso’s linear interpolation inference as described above results in a continuous transition between two keyframes, but not all responsive UI behavior can be represented in this way; some responsive behaviors require consistent properties within a range and discontinuous jumps between ranges. Espresso lets the user encode discontinuous jumps in element property behavior between two adjacent keyframes k_i and k_{i+1} . The location at which the discontinuity occurs affects the behavior for the range of viewport widths $w \in [w_i, w_{i+1}]$.

As a way of specifying the discontinuity, the Expresso user interface allows the user to set the behavior of an element property between two keyframes. For the range $r_{i,i+1}$ between k_i and k_{i+1} , the behavior could be:

- the linear interpolation behavior between k_i and k_{i+1} (which is the default) (Fig. 3, left),
- continued linear behavior from smaller viewport widths (range $r_{i-1,i}$ between k_{i-1} and k_i) (Fig. 3, middle), or
- continued linear behavior from larger viewport widths (range $r_{i+1,i+2}$ between k_{i+1} and k_{i+2}) (Fig. 3, right).

These three behaviors are illustrated in Fig. 3¹. In this paper, we refer to the first transition type (linear interpolation) as being a “smooth” transition and the second two transition types as being “jump” transitions. For example, the laptop in Fig. 1 should jump in y-position between keyframe k_1 (of width $w_1 = 990$ pixels) and keyframe k_2 (of width $w_2 = 1000$ pixels), as there is a major layout rearrangement between these two keyframes. Thus, the user uses the transition rules menu to specify that for range $r_{1,2}$, the laptop y-position should be that of an adjacent viewport range. In this case they choose for the laptop y-position in $r_{1,2}$ to be that of narrower viewports (i.e., range $r_{0,1}$), where the laptop is at the bottom of the page. This is indicated in the figure as the magenta jump transition, and this viewport range between 990 and 1000 pixels is contained in the region labeled “laptop is centered in viewport”.

¹If the continued behavior from range $r_{i-1,i}$ is chosen for $r_{i,i+1}$, but there is already a discontinuity between $r_{i-1,i}$ and k_i , then the behavior for range $r_{i,i+1}$ will just be the constant value specified at keyframe k_i .

E. Rule Representation

As discussed above, in Expresso the behavior of an element property over a viewport size range takes the form of a linear equation. Whether the behavior over a range is a linear interpolation or is continuing that of an adjacent range, the behavior will be linear. Therefore, Expresso represents each element property behavior as a piecewise function, with a sub-function $propertyValue = mw + b$ defined for the range between each pair of adjacent keyframes.

F. Scope of Supported Behaviors

1) *Single Dimension Dependent:* In our examples with Expresso, we limit responsive behavior to be dependent on only one viewport dimension: width. We chose to support responsive behavior with respect to viewport width because we observed that responsive UIs most often react to changes in viewport width, as vertically scrolling a website to view more content is common. Supporting responsive behaviors dependent on one variable only (e.g., viewport width, viewport height, or scroll position) is straightforward, requiring only a first-order polynomial (which Expresso already supports) for fit. To support responsive behavior for a given element property dependent on both viewport width and height would require a higher-order polynomial to be fully expressive.

2) *Types of Transitions:* In the current implementation, we limit the kinds of transitions to smooth linear interpolation and discontinuous jumps. Other responsive behaviors can be supported using our approach (e.g., quadratic or exponential relationships), but for Expresso we chose linear slopes and jumps since these support continuous and discontinuous transitions, respectively. Future versions of our tool could support more transition behaviors to suit additional use cases.

3) *Types of Properties:* Expresso currently supports specifying x-position, y-position, width, height, font size, text color, and background color in keyframes. In the future we plan to explore adding other properties to Expresso. For example, many responsive UIs change elements’ visibility depending on viewport size to hide or swap out elements. This would essentially be a degenerate case of the current linear equation and discontinuity representation: a UI element’s “visibility” attribute would be either “visible” or “hidden” for each continuous range.

G. Implementation

Expresso is implemented as a Node.js web application. Raw data about property values for each element for each keyframe are stored in a JavaScript object, which is updated as the user adds keyframes, modifies elements in the UI, and sets transition metadata. An initial, static web page can be loaded into Expresso as a JavaScript Object Notation (JSON) object containing one keyframe. As the user makes updates to their keyframes, Expresso recomputes a piecewise function per element property as explained in the “Transition Behaviors” section above. When the user resizes the page viewport, Expresso updates element CSS property values according to the piecewise functions. Currently, Expresso uses JavaScript

to update CSS property values rather than generating dynamic CSS. Future versions could instead generate responsive CSS and relationships via `calc` and the viewport `vw` unit.

Note that elements and their property values in Espresso are represented as a flat hierarchy. Currently there is no notion of elements belonging to a common parent container. Raw element position values in the JavaScript object are relative to the top-left corner of the web page viewport container. Elements are therefore absolutely positioned, independent of each others' positions. Future versions of Espresso could potentially represent elements in a hierarchical manner to better match typical HTML structure, especially if we support importing existing code.

V. EVALUATION

We conducted a laboratory study to evaluate whether Espresso can help individuals with minimal CSS experience to build responsive UIs. In our study, we asked participants to use Espresso to build two responsive web pages, for which we provided visual specifications.

A. Participants

We recruited six participants² (two female and four male, $\mu = 22.3$ years old, $\sigma = 3.35$ years) with minimal CSS experience. Two participants reported over five years of general programming experience, three participants reported 2–5 years, and one participant reported 1–2 years. All participants reported one year or less of CSS experience; four participants reported a week or less, one reported 2–4 weeks, and one reported 3–6 months.

B. Study Design

The primary goal of our study was to determine how feasible it is for users — particularly those with minimal CSS experience — to build responsive web pages using Espresso. We first gave participants a tutorial of Espresso and then presented them with two responsive web page building tasks to learn how feasible the tool was to use for a variety of responsive behaviors.

1) Tutorial: We gave participants a 15 minute tutorial at the beginning of each session to familiarize the participant with the features of Espresso. In this tutorial, we showed participants an example responsive web page at different stages of its development in Espresso, demonstrating how to achieve different responsive behaviors. In particular, we explained the concept of transitions between two keyframes and how to encode “smooth” and “jump” transitions.

2) Tasks: We presented participants with two tasks each, for which we counterbalanced the order. Each task had two smooth transitions and one jump transition. For each task, participants were given a starter web page with one keyframe (therefore no responsive behavior) and a set of GIFs demonstrating the desired responsive behavior for the web page. Participants were shown four GIFs per task: one GIF illustrating the overall responsive behavior, and three GIFs illustrating

the behavior of every transition (one GIF per transition). We used these broken down GIFs in order to help convey the behaviors that they should be building without providing clues about the solution. Participants were asked to encode the responsive behavior in Espresso and were instructed to inform the researcher when they felt they had completed the task or if they could no longer make progress.

We chose to use pre-determined tasks, as opposed to open-ended tasks (“make this static page responsive, however you see fit”), to allow us to better evaluate participants’ performance. With open-ended tasks, it would have been difficult to determine if a participant implemented a particular behavior because it was what they wanted or because it was easy.

3) Task web pages: The two web pages we chose for the study are adapted from real web pages, represent different layout styles, and include realistic behaviors. These responsive behaviors include: element resizing relative to the page width, element centering, flexible grid behavior, and arbitrary element rearrangement. The two tasks were:

- Task A: The Mozilla web page³ (Fig. 1), which consists of a laptop, white text, and a blue background. For wide page viewports, the top half of the page is filled with a blue background, and the text occupies the left side of the viewport and the laptop the right side of the viewport. The laptop remains centered in its blue area on the right. For narrower viewports, the full page height is filled with the blue background, and the text and laptop are stacked vertically and horizontally centered. Each of these two layouts therefore has smooth transition behavior. At a viewport width of 1000 pixels, the layout immediately jumps from one layout to the other.
- Task B: The Bass web page⁴ (Fig. 2), which consists of a set of six shoes, a brown banner with “Bass” text, and a left menu. The brown Bass banner always appears at the top of the page with the “Bass” text horizontally centered. For wide page viewports, the six shoes appear in a 3×2 grid, with the shoes shrinking in size and becoming closer together as the page narrows. The left menu also shrinks in width as the page becomes narrower. For narrower viewports, the six shoes appear in a 2×3 grid, with the shoes initially large and then shrinking, and the left menu has a constant width. At a viewport width of 780 pixels, the layout immediately jumps from one layout to the other, resulting in an immediate jump from the 3×2 to 2×3 grid, as the transition widget in Fig. 2c shows.

C. Results

We evaluated the web pages participants created in Espresso against the same rubric we used in the motivational CSS study. Elements that shared the same kind of behavior (e.g., all of the

³Adapted from <https://web.archive.org/web/20180428062643/https://www.mozilla.org/en-US/>

⁴Adapted from <https://web.archive.org/web/20170928121043/https://www.ghbass.com/category/g+h+bass/weejuns/women.do>

²There was no overlap in participants with the motivational study.

Statement	Mean rating (1 to 7)	Standard deviation
<i>Using this tool in my job would enable me to accomplish tasks more quickly.</i>	6.33	0.471
<i>Using this tool would improve my job performance.</i>	5.67	0.745
<i>Using this tool would enhance my effectiveness on the job.</i>	5.83	0.898
<i>Using this tool would make it easier to do my job.</i>	6.17	0.373
<i>I would find this tool useful in my job.</i>	6.17	0.373
<i>Learning to operate this tool would be easy for me.</i>	6.67	0.745
<i>I would find it easy to get this tool to do what I want it to do.</i>	5.50	0.957
<i>My interaction with this tool would be clear and understandable.</i>	6.33	1.11
<i>I would find this tool to be flexible to interact with.</i>	6.17	1.07
<i>It would be easy for me to become skillful at using this tool.</i>	6.33	0.745
<i>I would find this tool easy to use.</i>	6.67	0.471

TABLE I: Results of the Technology Acceptance Model (TAM) questionnaire we presented participants, with each statement rated on a scale from 1 (extremely unlikely) to 7 (extremely likely).

white text in the Mozilla example were either all left-aligned or center-aligned), fell under one rubric item. Note that we evaluated accuracy of tasks by reviewing work completed by the 22.7 minute mark. We retroactively chose this cutoff time based on the earliest time we asked a participant to end their work before they had finished. For the Mozilla task (Fig. 1), participants achieved a mean accuracy of 80.7% ($\sigma = 15.9\%$), with a mean completion time of 12.5 minutes ($\sigma = 4.95$ m). For the Bass task (Fig. 2), participants achieved a mean accuracy of 72.2% ($\sigma = 24.6\%$), with a mean completion time of 17.3 minutes ($\sigma = 2.87$ m). Overall, participants achieved a mean accuracy of 76.5% ($\sigma = 21.2\%$), with a mean completion time of 14.9 minutes ($\sigma = 4.70$ m).

After participants completed their tasks, we asked them to complete a TAM questionnaire, with each statement to be rated on a scale from 1 (extremely unlikely) to 7 (extremely likely). When presented with the statement “I would find this tool useful in my job”, participants responded with a mean rating of 6.17 ($\sigma = 0.373$). When presented with the statement “I would find this tool easy to use”, participants responded with a mean rating of 6.67 ($\sigma = 0.471$). Average results for the full set of TAM statements are reported in Table I.

We also conducted a short interview with participants to better understand their experience using Expresso and how it compared to other user interface building tools they had used. Most participants expressed satisfaction with Expresso, finding that it was easier to use than CSS while also supporting greater customizability than other tools they used:

P2: “*If I was making a website where I wanted custom control of how all the elements bounced around and I didn’t want to constrain myself to some given library that did it all automatically, then I would use this tool...*”

P4: “[“How does your experience using Expresso compare to your experience using other tools?”] This is definitely much easier. Because with templates, sometimes I will want to add new functionality to that. When that happens, it becomes much more complicated, because I need to first find example code online, how to do that, and then I need to copy that code into my template and debug to make it

work for the current template.

Participants also generally commented positively on the keyframe and transition paradigm that Expresso uses:

P1: “*In general, just thinking about how you can break up something that has complex behavior into a single keyframe is beneficial because you don’t have to worry about everything at once, you can kind of focus on one aspect... Getting the first animation working was fluid and quick because you just start somewhere and end somewhere and you just specify what kind of transition you want.”*

P2: “*The idea of using keyframes seemed very intuitive to me because I’ve used that sort of design with video editing and animations.”*

However, participants did also experience some challenges when using keyframes:

P1: “*When I was using the tool...I found it kind of hard to think about the different stages of my UI in terms of keyframes.”*

P3: “*If I didn’t think about keyframes I actually needed, it became more difficult as I tried to add keyframes later on... I think it would be easier for me if I actually thought about what I was doing first, like making an outline.”*

P5: “*The difficulty was how to select the keyframes. You need to pick out the keyframes at the right time...if you want to shrink smoothly and then suddenly change from 3 columns to 2 columns, in fact you need to insert 2 keyframes here like with similar pixels, but at the beginning I didn’t know that because I was not familiar with this pipeline. I only inserted 1 keyframe and found that it was unable to do the job, so I noticed I needed another one.”*

Participants also expressed desire for authoring features common in commercial products to make the tool more usable, in particular element alignment, snapping, and centering.

We also observed some interesting usage patterns.

1) *Keyframes straddling “jump” were often close in size:* All six participants, in at least one task each, placed their two

middle keyframes (surrounding the expected “jump” transition) close to each other. The difference in viewport size of the two keyframes was 23 pixels or less in 10 of 12 trials, and was 3 pixels or less in 6 of 12 trials. There are a couple possible reasons for this pattern. In the Espresso tutorial example we presented, there was a 4 pixel difference in viewport size for the two keyframes surrounding the jump transition, so maybe this biased the participants. However, perhaps participants found a small range between the two layout specifications to be advantageous, to have more control over the UI behavior in this viewport size range, or to minimize the viewport range affected by a “jump” transition (which was still a new concept to participants).

2) Trouble when keyframes straddling “smooth” were close in size: In a couple cases each, two participants created keyframes very close in viewport size that straddled a “smooth” transition. They then created significant UI element position and size changes between the adjacent keyframes, which they later realized were not appropriately proportionate to the change in viewport size. When they resized the viewport for testing, side effects included shoes shrinking or growing too quickly (quickly becoming minuscule or taking up the full viewport), or elements flying off the page. In the future perhaps Espresso could warn users when it notices a large UI element property change over a small range, or Espresso could support modifying a keyframe’s viewport size after it’s been created (i.e., to move the two keyframes further apart).

D. Discussion

As reported in their TAM scores and interviews, participants generally found Espresso to be useful and easy to use. This improvement in self-efficacy can help engage non-programmers in technical problem solving and potentially be usable as a scaffold for teaching computing and programming concepts to non-experts [37].

Although participants reported high TAM scores for Espresso (see Table I), the web pages they built were not perfect according to our rubric (e.g., 76.5% overall accuracy). However, these accuracy scores represent a lower-bound on participants’ ability to use Espresso because the error rate includes not only user mistakes in using the tool, but also errors in user intent due to most participants overlooking some aspect of system behavior in the GIFs. Since the participants saw the GIFs for the first time during the task and did not design the web pages and behaviors themselves, they first needed to interpret the behaviors in the GIFs before encoding them with Espresso. One example of a commonly missed behavior was the somewhat subtle shrinking of the left menu in the Bass task.

The relatively high TAM scores indicate that participants found the tool easy to use and useful. This also suggests that participants mostly built what they intended to, even if they misinterpreted the behavior specified by the instructional GIFs. This appeared to be the case from the recordings: when users attempted to demonstrate a behavior, they generally succeeded, and most of the failures we recorded appeared to be due to not taking any intentional steps towards adding it. Since we

envision real users to be individuals who already know what specific responsive behaviors they want their user interface to have, the ability to use Espresso to encode intended behaviors is the most relevant success measure. Further, it suggests that understanding and communicating system state and current behavior is a key need for supporting non-programmers. We discuss this further in the next section.

VI. FUTURE WORK

Participants were generally successful encoding the necessary transitions into Espresso to complete their tasks, but did not always encode them correctly on their first try, or took time to determine which dropdown menu item they needed to select in order to achieve the desired discontinuity behavior. Future work may explore how to devise and evaluate visualizations to help users better understand the current global behavior of elements across the state space and plan for future modifications. The visualization should also be interactive to support some of these behavior modifications.

Also, as mentioned in the “Scope of Supported Behaviors” section, our keyframe and transition approach could be adjusted to support building web pages that are responsive in both their viewport width and height. One approach would be to use a system of equations with higher-order polynomials (e.g., quadratic functions) to calculate element behavior definitions that satisfy all keyframes in two-dimensional space. This would require additional demonstrations to fully specify. Alternatively, some websites’ responsive behavior should be strict per dimension, regardless of the other dimension’s value. Supporting separate rules per dimension could be beneficial, but would need to be designed such that conflicts between viewport width and height rules are avoided or easily fixed.

VII. CONCLUSION

In this paper, we introduced Espresso, a system for creating responsive UIs by specifying keyframes over a UI property (e.g., page width) and setting transitions between them. These keyframes and transitions are used to generate responsive layout rules. We found that even individuals with little to no CSS experience are able to specify complex responsive UIs with Espresso, achieving a mean accuracy of 76.5% in their tasks, and rating it highly on the TAM scale as useful and easy to use. Meanwhile, individuals with similar experience who tried to build these same responsive UIs using CSS were much less successful. More broadly, our work takes a step toward a future in which users can provide intuitive demonstrations to guide the automatic creation of complex UI behaviors.

VIII. ACKNOWLEDGEMENTS

We thank Yan Chen and Stephanie O’Keefe for their help editing this paper; Jordan Huffaker, Xiaoying Pu, and Kayla Wiggins for their feedback on the Espresso UI; and our study participants for their time and effort. This work was supported in part by Clinec, Inc., and the University of Michigan.

REFERENCES

- [1] H. S. Liang, K. H. Kuo, P. W. Lee, Y. C. Chan, Y. C. Lin, and M. Y. Chen, "Seess: seeing what i broke—visualizing change impact of cascading style sheets (css)," in *Proceedings of the 26th annual ACM symposium on User interface software and technology*. ACM, 2013, pp. 353–356.
- [2] D. Mazinanian, "Refactoring and migration of cascading style sheets: Towards optimization and improved maintainability," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 1057–1059. [Online]. Available: <http://doi.acm.org/10.1145/2950290.2983943>
- [3] N. Burtnyk and M. Wein, "Computer-generated key-frame animation," *Journal of the SMPTE*, vol. 80, no. 3, pp. 149–153, 1971.
- [4] B. Myers, S. E. Hudson, and R. Pausch, "Past, present, and future of user interface software tools," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 7, no. 1, pp. 3–28, 2000.
- [5] Apple, Inc. (2003) Xcode. [Online]. Available: <https://developer.apple.com/xcode/>
- [6] Google, Inc. (2013) Android studio. [Online]. Available: <https://developer.android.com/studio/index.html>
- [7] S. Oney, B. Myers, and J. Brandt, "Constraintjs: programming interactive behaviors for the web by integrating constraints and states," in *Proceedings of the 25th annual ACM symposium on User interface software and technology*. ACM, 2012, pp. 229–238.
- [8] S. Oney, B. Myers, and J. Brandt, "Interstate: a language and environment for expressing interface behavior," in *Proceedings of the 27th annual ACM symposium on User interface software and technology*. ACM, 2014, pp. 263–272.
- [9] D. A. Henderson Jr, "The trillium user interface design environment," *ACM SIGCHI Bulletin*, vol. 17, no. 4, pp. 221–227, 1986.
- [10] Adobe Systems. (1997) Dreamweaver. [Online]. Available: <https://www.adobe.com/ca/products/dreamweaver.html>
- [11] Webflow, Inc. (2013) Webflow. [Online]. Available: <https://webflow.com/>
- [12] Zine EOOD. (2016) Bootstrap studio. [Online]. Available: <https://bootstrapstudio.io/>
- [13] B. A. Myers, "Peridot: creating user interfaces by demonstration," in *Watch what I do*. MIT Press, 1993, pp. 125–153.
- [14] D. Kurlander and S. Feiner, "Inferring constraints from multiple snapshots," *ACM Transactions on Graphics (TOG)*, vol. 12, no. 4, pp. 277–304, 1993.
- [15] A. Repenning and T. Sumner, "Agentsheets: A medium for creating domain-oriented visual languages," *Computer*, vol. 28, no. 3, pp. 17–25, 1995.
- [16] H. Lieberman, "Tinker: A programming by demonstration system for beginning programmers," *Watch what I do: programming by demonstration*, vol. 1, pp. 49–64, 1993.
- [17] H. Lieberman, "Mondrian: a teachable graphical editor." in *INTERCHI*, 1993, p. 144.
- [18] T. Lau, L. Bergman, V. Castelli, and D. Oblinger, "Sheepdog: learning procedures for technical support," in *Proceedings of the 9th international conference on Intelligent user interfaces*. ACM, 2004, pp. 109–116.
- [19] A. Cypher, "Eager: Programming repetitive tasks by demonstration," in *Watch what I do*. MIT Press, 1993, pp. 205–217.
- [20] A. F. Blackwell, "Swyn: A visual representation for regular expressions," in *Your wish is my command*. Elsevier, 2001, pp. 245–XIII.
- [21] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld, "Learning repetitive text-editing procedures with smartedit," in *Your wish is my command*. Elsevier, 2001, pp. 209–XI.
- [22] R. C. Miller and B. A. Myers, "Multiple selections in smart text editing," in *Proceedings of the 7th international conference on Intelligent user interfaces*. ACM, 2002, pp. 103–110.
- [23] H. Lieberman, *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.
- [24] A. Cypher and D. C. Halbert, *Watch what I do: programming by demonstration*. MIT press, 1993.
- [25] M. R. Frank, P. N. Sukaviriya, and J. D. Foley, "Inference bear: designing interactive interfaces through before and after snapshots," in *Proceedings of the 1st conference on Designing interactive systems: processes, practices, methods, & techniques*. ACM, 1995, pp. 167–175.
- [26] Y. Li and J. A. Landay, "Informal prototyping of continuous graphical interactions by demonstration," in *Proceedings of the 18th annual ACM symposium on User interface software and technology*. ACM, 2005, pp. 221–230.
- [27] M. Bolin, M. Webber, P. Rha, T. Wilson, and R. C. Miller, "Automation and customization of rendered web pages," in *Proceedings of the 18th annual ACM symposium on User interface software and technology*. ACM, 2005, pp. 163–172.
- [28] R. C. Miller, V. H. Chou, M. Bernstein, G. Little, M. Van Kleek, D. Karger *et al.*, "Inky: a sloppy command line for the web with rich visual feedback," in *Proceedings of the 21st annual ACM symposium on User interface software and technology*. ACM, 2008, pp. 131–140.
- [29] G. Little, T. A. Lau, A. Cypher, J. Lin, E. M. Haber, and E. Kandogan, "Koala: capture, share, automate, personalize business processes on the web," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2007, pp. 943–946.
- [30] G. Leshed, E. M. Haber, T. Matthews, and T. Lau, "Coscripter: automating & sharing how-to knowledge in the enterprise," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2008, pp. 1719–1728.
- [31] S. Oney, A. Lundgard, R. Krosnick, M. Nebeling, and W. S. Lasecki, "Arboretum and arbority: Improving web accessibility through a shared browsing architecture," in *Proceedings of the ACM Symposium on User Interface Software and Technology*. ACM, 2018.
- [32] W. S. Lasecki, J. Kim, N. Raftor, O. Sen, J. P. Bigham, and M. S. Bernstein, "Apparition: Crowdsourced user interfaces that come to life as you sketch them," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM, 2015, pp. 1925–1934.
- [33] S. W. Lee, Y. Zhang, I. Wong, Y. Y., S. O'Keefe, and W. Lasecki, "Sketchexpress: Remixing animations for more effective crowd-powered prototyping of interactive interfaces," in *Proceedings of the ACM Symposium on User Interface Software and Technology*, ser. UIST. ACM, 2017. [Online]. Available: <https://doi.org/10.1145/3126594.3126595>
- [34] J. Nichols, B. A. Myers, M. Higgins, J. Hughes, T. K. Harris, R. Rosenfeld, and M. Pignol, "Generating remote control interfaces for complex appliances," in *Proceedings of the 15th annual ACM symposium on User interface software and technology*. ACM, 2002, pp. 161–170.
- [35] K. Gajos and D. S. Weld, "Supple: automatically generating user interfaces," in *Proceedings of the 9th international conference on Intelligent user interfaces*. ACM, 2004, pp. 93–100.
- [36] B. Myers, S. Y. Park, Y. Nakano, G. Mueller, and A. Ko, "How designers design and program interactive behaviors," in *Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008. IEEE Symposium on*. IEEE, 2008, pp. 177–184.
- [37] D. Loksa, A. Ko, W. Jernigan, A. Oleson, C. J. Mendez, and M. M. Burnett, "Programming, problem solving, and self-awareness: Effects of explicit guidance," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '16. ACM, 2016, pp. 1449–1461.

The design and evaluation of a gestural keyboard for entering programming code on mobile devices

Gennaro Costagliola, Vittorio Fuccella, Amedeo Leo, Luigi Lomasto, Simone Romano

Department of Informatics

University of Salerno

Fisciano (SA), Italy

Email: {gcostagliola,vfuccella}@unisa.it

Abstract—We present the design and the evaluation of a soft keyboard aimed at facilitating the input of programming code on mobile devices equipped with touch screens, such as tablets and smartphones. Besides the traditional *tap on a key* interaction, the keyboard allows the user to draw gestures on top of it. The gestures correspond to shortcuts to enter programming statements/constructs or to activate specific keyboard sub-layouts. The keyboard was compared in a user study to a traditional soft keyboard with a QWERTY layout and to another state-of-art keyboard designed for programming. The results show a significant advantage for our design in terms of speed and gesture per characters.

Index Terms—Virtual keyboards; Gesture-based interaction; Touch screens.

I. INTRODUCTION

In recent years mobile devices (especially smartphones and tablets) replaced the traditional personal computers in many situations and for many applications. Thus, writing and editing programming code directly with one of these devices became a realistic possibility. Several researchers are convinced that computer programming will occur directly on mobile devices in the future: in the recent literature [1], [2], [3] as well as in the market of applications, there are a few proposals to facilitate the task of entering programming code on mobile devices equipped with touch screens.

Besides the new possibilities that the future will open up, there are already some scenarios where it is realistic to think that programming may even occur on a very small device, such as a smartphone:

- in many countries, commuters travel on crowded transport where it may be impossible to use a laptop or a tablet. Programmers in this situation can turn downtime into activity by editing code, e.g., by adding small features to their programs;
- programmers who receive calls requesting urgent software correction can perform this task on their smartphones when they do not have the immediate availability of their laptops;
- according to some authors (e.g., Tillman et al. [4]), due to their wide availability to users, most of the e-learning activities in the future will be carried out by students on smartphones. On these devices, they could execute homeworks consisting of small programming tasks.

Entering text with touch screens has always been a difficult task, both because of the reduced size of the screens and due to the lack of tactile feedback [5]. To facilitate text entry several solutions were proposed, such as the use of optimized layouts alternative to QWERTY [6], [7], [8], [9] and gestural input methods [10], [11].

Entering programming code with touch screens offers additional challenges. In particular, the programming code has often a large number of non-alphanumeric symbols that would force the user to frequently switch between different keyboard layouts. However, the narrow vocabulary and the regularities that can be found in many programming languages offer some cues and the groundwork for major improvements in code writing speed and comfort. Some tools were already proposed to improve editing operations of the programming code, such as *TouchDevelop* [4], while there are only a few designed for the direct input of programming code.

Here we present a gestural soft keyboard designed to improve the entry of programming code. It works as a traditional soft keyboard enabling the *tap on a key* interaction. Additionally, the keyboard allows the user to draw gestures on top of it. Each gesture is associated to a specific action which can be either:

- the entry of a programming statement/construct, or
- the entry of a symbol, or
- the activation of a sub-layout containing groups of logically related keys.

Several previous researches inspired our design. In particular, the idea of issuing commands by drawing gestures directly on the keyboard area, so as not to overlap the standard functionality of the keyboard or occupy the space of application gestures, is derived from Fuccella et al. [12]. The insertion of symbols in a similar way is introduced in [13]. The use of keyboard shortcuts to access sub-layouts or enter programming statements is derived from [1]. The problem of having to frequently switch between different keyboard layouts to enter symbols was also identified by Ihantola et al. [14], who also recommended the use of gestures to invoke editor commands.

In the design of a gesture-based application, several challenges arise. Bearing in mind that our objective is to improve the entry of source code on touch screens with respect to speed, accuracy and comfort, the main challenges were:

- the choice of the set of gestures to associate to frequent programming constructs. The set must have a limited size and gestures must be easy to execute and remember. Furthermore, they must be distinctive to each other to reduce ambiguity, which would result in recognition errors. Our solutions are described in Section III-A;
- gesture recognition. Despite the availability of good recognizers [15], [16], the chosen recognizer must be tuned by selecting the right templates and their number, and by setting its parameters. Our solutions are described in Section III-B.

The keyboard was evaluated in a user study where we measured its performance in terms of typing speed, accuracy and strokes per character. We also measured the perceived workload and user satisfaction through questionnaires. We compared our design to a traditional soft keyboard with a QWERTY layout and to the keyboard proposed by Almusaly and Metoyer [1], specifically designed for programming.

II. BACKGROUND

In this section, we survey some approaches for facilitating programming tasks on touch screens and give some basic information about the methods to evaluate text entry systems on mobile devices.

A. Programming on Touch Screens

In the literature we can find some proposals to improve the entry of programming code with touch screens, but they are mostly approaches to improve the editing of existing code. These include two gesture-based tools for refactoring code: Raab et al.'s *RefactorPad* [2] and the integrated design environment (IDE) proposed by Biegel et al. [3]. The main difference between them is that the former is designed to be operated through both finger and pen unistroke gestures, while the latter employs multi touch finger-based gestures.

The authors of RefactorPad [2], using the approach proposed in [17], establish a mapping between gestures and editor actions and propose some design guidelines for creators of code editors who wish to optimize their tools for touch screens. Biegel et al. [3] propose some guidelines for *touchifying* an IDE. These, besides the use of gestures to invoke refactoring commands, also include the re-design of menus, by replacing the menu of the Eclipse IDE with *radial* and *italic* menus to improve their use through fingers. *CodePad* [18] can be regarded as a precursor of the two above-mentioned tools. It proposes the use of interactive spaces on secondary multi-touch enabled devices (connected to a main one) to perform personal and collaborative programming tasks, including refactoring, visualization and navigation.

Other proposals include finger-based interaction techniques aimed at the improvement of IDEs for mobile devices. These are mostly widget-based interfaces. Hesenius et al. [19] proposed a tool to support the development of a concatenative language on touch screens. Their interface supports the insertion of keywords using drag-and-drop from a visual dictionary, the use of a toolbar to insert symbols and a mechanism

to interactively navigate through the code. McDirmid [20] proposed an interactive environment to enter code for the *YinYang* tile-based object-oriented programming language. The language itself is composed of interactive elements, mainly buttons and context menus, which can be tapped to perform operations, such as the addition of a new tile.

With *TouchDevelop* [4] Tillmann et al. propose a language and an IDE to help the user in the development of programs. The IDE includes a semi-structured editor and a domain-specific soft keyboard to edit expressions.

Almusaly and Metoyer [1] introduced a soft keyboard which uses a syntax-directed approach to help the user in entering syntactically correct code. Their keyboard has a primary layout containing shortcuts for entering programming keywords and constructs. Furthermore, various sub-layouts can be activated for specific functions. Keys for "free" typing are present in a secondary QWERTY layout, which is shown by pressing a key in the primary layout.

In a more recent paper [21], the same authors enhanced their previous design and performed a longitudinal study (spanning 8 sessions) showing its learning curve. However, in this study they did not compare it with other input methods.

Besides research prototypes, there are commercial solutions in the form of applications downloadable from the online markets of the various mobile OSs. Among the soft keyboards specifically designed for programming we can mention *The Hacker's Keyboard*¹ for the Android system and *Textastic*² for iPad/iPhone. The former contains useful buttons, e.g. arrow keys and symbols, directly in the main layout. The latter has an upper key row containing many symbols which can be entered through taps or swipe gestures.

The approach more similar to ours is that of Almusaly and Metoyer [1]. Our criticism to their method is that it completely replaces the traditional QWERTY layout with a new layout composed of shortcut keys. This compels the programmer to perform frequent switches between layouts to switch from the insertion of constructs/keywords to free typing and vice versa. In our design, we always show both character and shortcut keys, enabling free writing and keyword entry seamlessly, i.e. without switching between layouts. This is possible without taking up too much space since part of the shortcut keys are replaced with suitable gestures.

B. Text Entry Metrics for Soft Keyboards

The text entry methods are primarily evaluated by two metrics: speed and accuracy. The evaluation is carried out mostly in empirical tests in which participants must transcribe short phrases from standardized sets, such as that proposed by MacKenzie and Soukoreff [22].

As for speed, the most used metric is *words per minute* (wpm): speed is calculated by dividing the total number of entered characters by the time to enter them. A *word* is conventionally composed of five characters [23].

¹<http://code.google.com/p/hackerskeyboard/>

²<https://www.textasticapp.com/>

The accuracy is evaluated by measuring the *error rate*, expressed as the percentage of incorrect characters w.r.t. the length of the text. The count of incorrect characters is based on the *minimum string distance* (MSD) between the *presented* text and the *transcribed* text. There are various metrics, but the most commonly used measure all the errors made while typing (Total Error Rate - TER) and the errors left in the transcribed text (Not Corrected Error Rate - NCER) [24].

A measure giving indication of both efficiency and correctness is the average number of keystrokes required to enter a single character - *KeyStrokes Per Character* (KSPC). In gestural methods, both gestures and taps are counted and the metric is referred to as *Gestures Per Character* (GPC) [25].

III. DESIGN AND IMPLEMENTATION OF THE KEYBOARD

In the following, we describe the operation of the keyboard along with a discussion on the major challenges faced in the design phase. The keyboard was designed for Java, but underlying idea can be valid for most programming languages.

Our interface is a soft keyboard with a QWERTY layout amended with the capability to interpret gestures drawn on top of it. The keyboard was implemented by customizing Android *Soft Keyboard*³ sample project. Besides the capability of using gestures, the keyboard differs from a traditional QWERTY soft keyboard since it has an additional row on top which displays a sub-layout. The sub-layout contains groups of keys which are shortcuts to enter code constructs and is activated using gestures. Regarding the set of constructs, we relied on the work already done in [1], which is based on the analysis of the frequency of the words and of the most common constructs of the Java language.

Figure 2a shows a mock-up of a gesture drawn (in red) on the top of the keyboard with the programming code produced in a note editor application as a result.

A. Gesture/Operation Mapping

Some gestures directly generate code while others activate sub-layouts containing a related set of keys. There are also gestures that do both actions, i.e., they both produce some code and activate a sub-layout.

The gestures-constructs mapping we used is shown in Table I. The fourteen gestures are all unistrokes. The second column of the table shows their shape. The gestures are drawn beginning from the dot. The third column specifies the type of gesture: gestures of type *Code* (C) directly produce code in output; gestures of type *Symbol* (S) produce a symbol; gestures of type *Sub-Layout* (SL) activate a sub-layout in the keyboard.

The sub-layouts are surrounded by differently coloured frames, to make the user aware of the currently activated sub-layout and of mode switching consequent to a gesture. The only gesture that produces both a code fragment and the activation of a sub-layout is gesture 10, which guides the user in the insertion of a new function.

We selected six sub-layouts, containing the following sets of keys:

³<https://developer.android.com/samples/index.html>

Number	Gesture	Type	Produced code/action
1		C	Comment delimiter
2		C	Class stub
3		S	'Greater than' symbol
4		S	'Lower than' symbol
5		C	main() method
6		C	println() method
7		C	Square brackets
8		C	Parentheses
9		SL	Access to <i>Iteration</i> sub-layout
10		C;SL	Function code; Access to <i>Function</i> sub-layout
11		SL	Access to <i>Control Operators</i> sub-layout
12		SL	Access to <i>Exception</i> sub-layout
13		SL	Access to <i>Structure</i> sub-layout
14		SL	Access to <i>Variable</i> sub-layout

TABLE I: The gesture-operation mapping used in our design. Each gesture was associated with action(s) from three categories: Code (C); Symbol (S); Sub-Layout (SL).

- 1) Gesture 9 (a circle) opens a sub-layout containing loop statements, that is: *for*, *while*, *do while*, *continue*, *break*;
- 2) Gesture 10 (an *f* character) produces a code fragment and shows a sub-layout containing the statements needed to write a function, that is: *modifiers*, *return type*, *rename*, *parameters*;
- 3) Gesture 11 (a question mark) shows a sub-layout with conditional statements, such as: *if*, *else*, *switch*, *case*, *break*, *default*;
- 4) Gesture 12 (an *e* character) opens the exception sub-layout: *try*, *catch*, *throw*;
- 5) Gesture 13 (an *S* character) shows a sub-layout with keywords to create collections: *Map*, *Hashmap*, *ArrayList*, *array*, *matrix*.
- 6) Gesture 14 (a *v* character) opens the variable sub-layout, containing: *modifiers*, *type*, *rename*, *assign*;

There are also two sub-sub-layouts, opened by pressing a sub-layout button:

- 1) A click on "type" in variable sub-layout or on "return type" in function sub-layout opens a sub-sub-layout containing common variable types, i.e. : *int*, *double*, *boolean*, *char*, *String*, *void*, *Custom*;
- 2) A click on "modifiers" in variable or function sub-

layout shows modifiers sub-sub-layout, containing: *private*, *public*, *protected*, *static*, *final*, *public static*.

We know from previous research (e.g. [12]) that positioning the cursor in the text on touch-screens is an inefficient operation. It is a good design choice to minimize the amount of such operations to speed up text entry. To this aim, we chose to automatically positioning the cursor in a convenient place when programming constructs are entered. Such a place is the one where the next text input is expected. For instance, if users want to specify parameters after they have entered the name of the function, they can simply click on the *parameters* key, located in the *Function* sub-layout (see Figure 2a); the cursor is automatically placed between the two parentheses.

B. Gesture Recognition

To classify gestures, we used the unistroke recognizer by Fuccella and Costagliola [15]. As rotation invariance was not required, the recognizer was instantiated in a rotation sensitive context. A simple threshold on the length of the stroke was used to distinguish gestures from simple taps. The threshold was set at the size of the smallest of the keys.

The choice of both the gestures described in the preceding subsection and the number of templates to use for each gesture was calibrated on the basis of the performance of the recognizer. In particular, our design passed through several iterations through which we made changes until we got an accurate result.

We tuned the recognizer in writer-independent tests with 8 participants. We gathered four samples for each gesture and for each participant to obtain a total number of 448 (4 samples \times 14 classes \times 8 participants) sample gestures. We tested the recognizer using a cross-validation: for each gesture class, we randomly chose the templates from n-1 participants and calculated recognition rates on the n-th participant. We performed 1000 trials for each of the n participants and then averaged the results them.

The recognition results using 3 samples per class are reported in the confusion matrix in Figure 1. We regarded as satisfactory such a result (about 1% error rate). Since the recognizer was also efficient, in our final configuration we chose to use 3 samples per gesture, taken at random from our gathered dataset. As we can see from the matrix, the lowest accuracy on a single class was about 94%, which was acceptable to us.

IV. EVALUATION

We designed a user-study whose objective was to compare the performance of our keyboard with those of the keyboard described in [1]. We also regarded useful to include the simple QWERTY layout as input method, since the authors in [1] failed in obtaining a faster entry with respect to such a baseline layout.

A. Participants

We recruited 15 participants (2 female) among students (graduated, undergraduate and Ph.D. candidates) in our university. Their ages ranged from 23 to 29 (M=25.2; SD=2.0).

Confusion Matrix															
Output Class	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Target Class
	800 7.1%	0 0.0%	0 0.0%	0 0.2%	22 0.0%	0 0.0%	24 0.0%	0 0.2%	26 0.0%	0 0.2%	0 0.0%	22 0.0%	0 0.0%	0 0.0%	89.5% 10.5%
	0 0.0%	800 7.1%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	100%
	0 0.0%	0 0.0%	800 7.1%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	100%
	0 0.0%	0 0.0%	0 0.0%	800 7.1%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	100%
	0 0.0%	0 0.0%	0 0.0%	0 0.0%	778 6.9%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	22 0.2%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	97.3% 2.7%
	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	800 7.1%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	100%
	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	800 7.1%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	100%
	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	776 6.9%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	100%
	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	800 7.1%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	100%
	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	752 6.7%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	100%
	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	800 7.1%	0 0.0%	0 0.0%	0 0.0%	100%
	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	778 6.9%	0 0.0%	0 0.0%	100%
	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	800 7.1%	0 0.0%	100%
100% 0.0%	100% 0.0%	100% 0.0%	100% 2.7%	97.3% 0.0%	100% 3.0%	97.0% 0.0%	100% 3.0%	94.0% 0.0%	100% 6.0%	97.3% 0.0%	100% 2.7%	100% 0.0%	99.0% 1.0%		

Fig. 1: The confusion matrix for three samples.

Participation was voluntary and participants were unpaid. All participants had previous experience with mobile devices and touch-screens. All of them had at least some programming experience. However, none of them had ever entered programming code on a smartphone.

B. Apparatus

The device used for the experiment was an *LG G3 D855 32GB* with a 5.5" display and a resolution of 2560 x 1440 pixels. The device ran the Android 5.0 Operating System.

The experimental software was composed of four modules: three soft keyboards and a text editor. The keyboards were all derived from the Android sample *Soft Keyboard* project and each of them was implemented as an independent Android service. In particular, the three keyboards were the following:

- *GesturalKeyboard (GK)*: the keyboard described in Section III (See Figure 2a).
- *Syntax-Directed Keyboard (SDK)*: the keyboard proposed in [1]. Due to the lack of the original implementation, we replicated the keyboard based on the description contained in their paper and on a video demonstration kindly provided by the authors. A picture of the replicated keyboard is shown in Figure 2b;
- *QWERTY*: the simple soft keyboard with the baseline layout, without any modification.

All the keyboards had size: 47.8mm \times 121.75mm on the device of the experiment.

Text editor: we developed a simple editor to allow our participants to carry out the two tasks; it was helpful to automatically calculate the output variables. The editor created

two log files for each task: a shorter file reporting the values of dependent variables (KSPC, WPM, TER, NCER) and a more detailed file containing all the events produced with user interactions, e.g. key presses, gestures, etc.

C. Procedure

Before starting the experiment each participant filled out a questionnaire with personal data and information on previous experiences related to the experiment. Then a training phase followed, where each participant practised with the two keyboards unfamiliar to him/her (Gestural Keyboard and Syntax-Directed Keyboard) for ten minutes each. During practice, participants were explained all of the shortcuts (gestures or key sequences) provided by the two keyboards and were invited to execute them. Participants were encouraged to ask any questions before the beginning of any task.

The experiment took place in a well-lit laboratory. The participant was sitting but free to adopt his/her preferred way to keep the device. Each participant carried out the experiment in three conditions (one for each keyboard). When operating with the gestural keyboard, the participants had the availability of a sheet showing the gestures and their corresponding operations. Each condition was composed of two tasks in which the participant had to copy a code block. The participant had to press a button to conclude the current task and to load the next one, if present. Between tasks, participants made a 2-minute break. Between test conditions they made a 5-minute break. Test conditions were counterbalanced among participants using a 3×3 latin square.

The code blocks for Task I and Task II are shown in Figures 3 and 4, respectively. The code was given to the participants printed on a sheet of paper. The syntax was suitably coloured as shown in the figures to improve readability. Here, in the figures (but not in the sheets given to participants), we underlined in red the text which had been entered character by character, i.e. which could not be entered using the shortcuts provided by the two specifically designed keyboards (GK and SDK). The tasks are very similar to those of the experiment described in [1].

The task duration, recorded to calculate the typing speed, was established as follows: it started when the user pressed his/her finger to enter the first character or to perform a gesture and ended when the user clicked on the "End task" button. Participants were allowed to correct errors while typing by using all means offered them by the keyboards (e.g. the backspace key). Furthermore, moving cursor interacting with the text editor was also allowed. Other interactions with the text editor, like selecting text and using the clipboard, were not allowed.

At the end of the experiment participants were asked to fill out a NASA Task Load Index (NASA-TLX) questionnaire to determine the mental demand, physical demand, time, performance, frustration and effort during the use of the three keyboards. The questionnaire was composed of statements to which the participants express their level of agreement in a 7

```
public class Point{
    private double x;
    private double y;
    // constructor
    public Point(double px, double py) {
        x = px;
        y = py;
    }
    public void setX(double px) {
        x = px;
    }
    public double getY() {
        return y;
    }
}
```

Fig. 3: The block of code to transcribe for Task I.

```
int[][][] matrix = new int[10][5];
for(int i=0; i<10; i++){
    for(int j=1; j<5; j++){
        System.out.println(i + j);
        if(i==j){
            System.out.println("diagonal");
        }
    }
}
```

Fig. 4: The block of code to transcribe for Task II.

levels Likert scale. Furthermore we collected some opinions and freeform comments.

D. Design

The experiment was a one-factor within-subjects design. The only tested factor was the *Input Method*, with the following three levels: GK; SDK and QWERTY.

The dependent variables were the *Speed* (in wpm); two variables to measure accuracy: TER and NCER; and the GPC. The above variables were calculated as specified in Section II-B. A participant's performance was obtained by averaging the results of the two tasks; the final value for a variable was calculated by averaging among participants.

V. RESULTS

The whole experiment lasted about one hour per participant. Our research hypothesis is that users can enter programming code faster, more accurately and with a smaller number of gestures with our Gestural Keyboard with respect to the baseline QWERTY layout and the keyboard described in [1]. Thus, our null hypotheses are that there are no differences between the corresponding measures of performance of the three different input methods. We used the ANOVA test to validate our results. For significant main effects, we used Sheffé post-hoc tests. The alpha level was set to 0.05.

A. Speed

The typing speeds are reported in Figure 5a. The grand mean for typing speed was 9.33 wpm. The fastest method

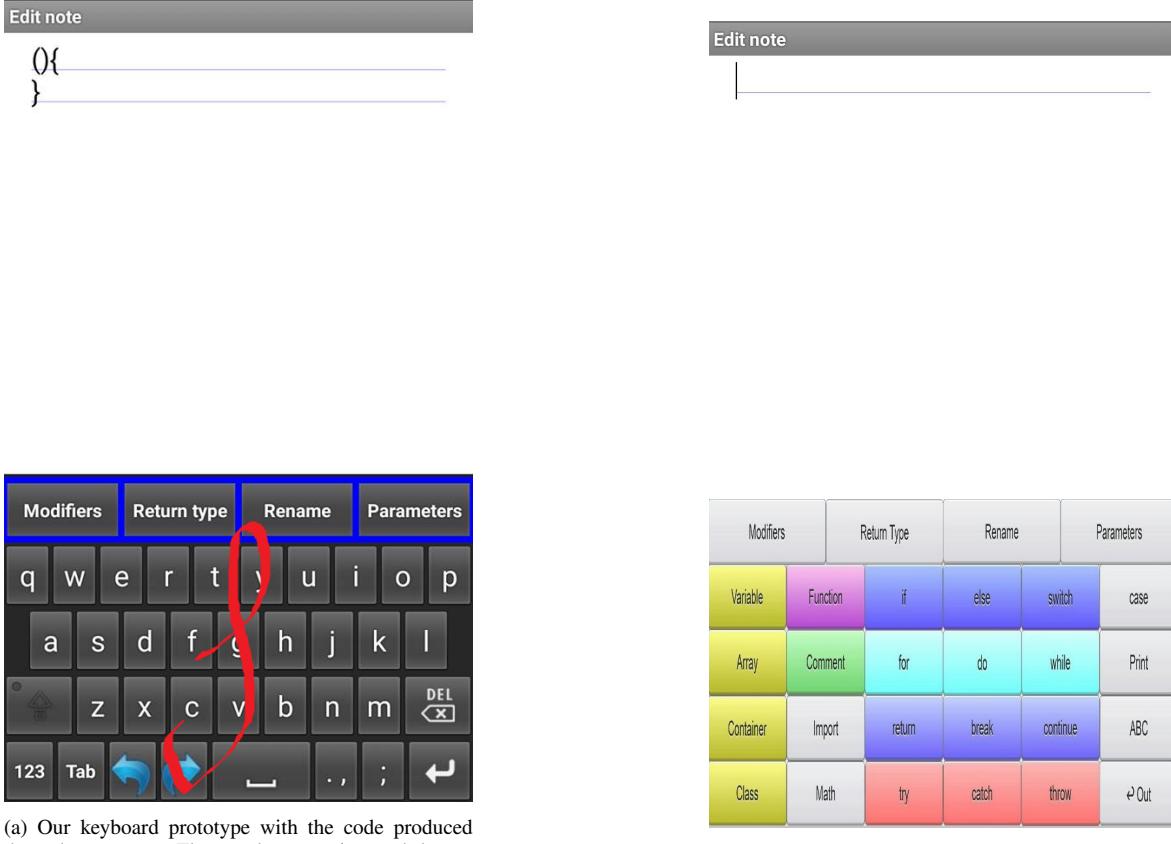


Fig. 2: The keyboards tested in the experiment.

TABLE II: Error rates obtained by the three input methods.

	GK	SDK	QWERTY
TER	12.22%	24.04%	14.11%
NCER	3.1%	6.23%	3.53%

was GK with 11.0 wpm, followed by QWERTY with 9.2 wpm and SDK with 7.7 wpm. From the ANOVA resulted that the main effect of the input method on the speed was statistically significant ($F_{2,28} = 13.89, p < .0001$). A Sheffé post-hoc analysis revealed that the significant difference was between GK and QWERTY and between GK and SDK.

B. Error Rate

Average values for TER and NCER are summarized in Table II. The grand mean for TER was 16.80%. The method with the lowest error rate was GK with 12.22%, followed by QWERTY with 14.11%, and SDK with 24.04%. From the ANOVA resulted that the main effect of the input method on the total error rate was statistically significant ($F_{2,28} = 604.67, p = .0443$). A Sheffé post-hoc analysis revealed that there was no significant difference between the means of the individual variables.

The grand mean for NCER was 4.28%. The method with the lowest error rate was again GK with 3.1%, followed

by QWERTY with 3.53% and SDK with 6.23%. From the ANOVA resulted no main effects of the input method on the NCER.

C. Gestures per character

The amounts of gestures per character for the three keyboards are reported in Figure 5b. The grand mean for GPC was 0.89. The method with the lowest gestures per character was GK with 0.61, followed by SDK with 0.79 and QWERTY with 1.27. From the ANOVA resulted that the main effect of the input method on the GPC was statistically significant ($F_{2,28} = 149.48, p < .0001$). A Sheffé post-hoc analysis revealed that the significant difference was between GK and SDK, GK and QWERTY, and between SDK and QWERTY.

D. Questionnaire results

The results of the NASA Task Load Index are shown in Figure 6. Overall, GK required less workload than the competing methods. In particular, it had lower average values for physical, time and effort and for frustration. The only value where QWERTY required less workload than GK is mental demand. This was probably due to the initial effort to learn the new method, while QWERTY was familiar to all participants. Furthermore, GK had a much higher result for the perceived performance. We analysed the results of the

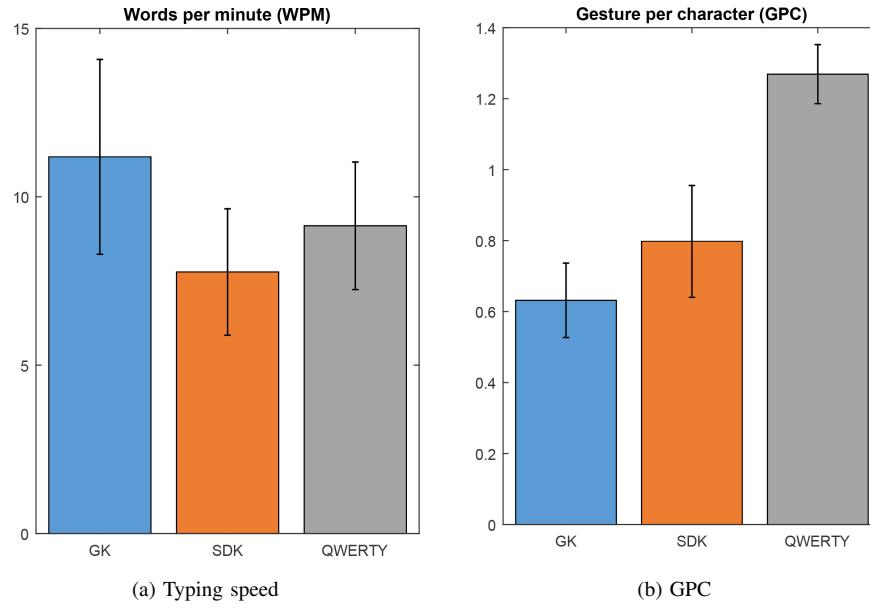


Fig. 5: Speed and GPC for the three tested input methods. One standard deviation error bars are shown.

questionnaire through a Friedman test with alpha level set at 0.05. For mental, physical, temporal demand and effort, we found a statistically significant difference between SDK and the other two input methods. For performance, we found a statistically significant difference between GK and the other two input methods. For frustration, we found a statistically significant difference between GK and SDK.

All participants except one chose the gestural keyboard as his/her favourite input method. When asked to express a judgement, participants stated they appreciated its high performance and simplicity. Some complaints were referred to gesture misinterpretation and to the lack of shortcuts for programming constructs (e.g., class constructors) and symbols (e.g., semicolon). As for SDK, some participants complained about the need of switching frequently between layouts and about the difficulty of learning a completely new layout. In general, the participants complained that the amount of practice was too short to make good use of the two novel keyboards.

VI. DISCUSSION AND CONCLUSION

We presented a gestural keyboard to improve the entry of programming code. The results of the user-study showed a clear advantage for our design over the compared input methods in terms of speed, comfort and user satisfaction.

Our keyboard can be adapted on differently sized touch screens. We preferred to use a mobile phone instead of a tablet (as in [1]) to test our keyboard in a more unfavourable setting (small screen). The slightly lower speeds obtained in our experiment are probably due to the smaller screen size.

An important difference in the results of our experiment and those described in [1] is the low user satisfaction with their keyboard. This result may be due to the presence in

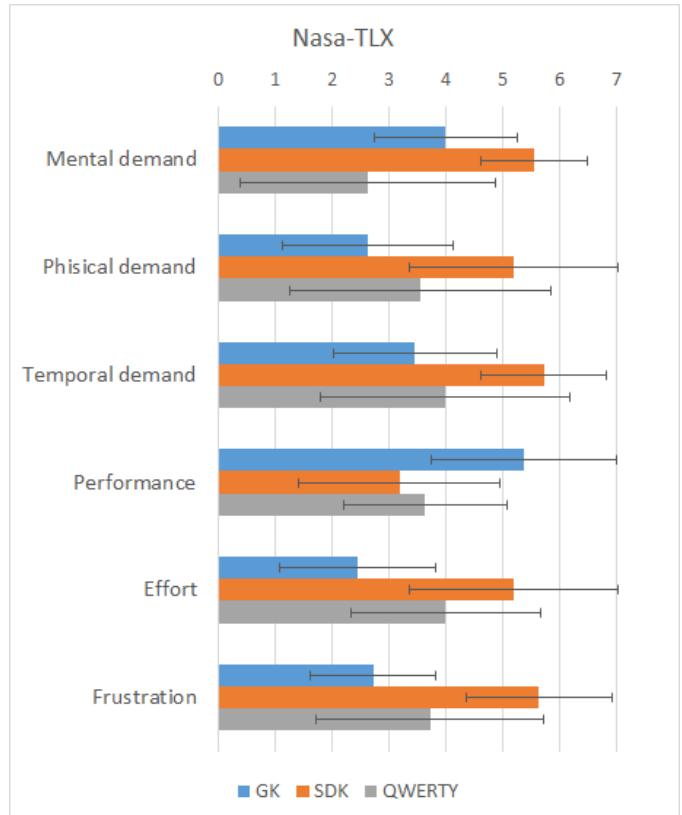


Fig. 6: Results of the Nasa-TLX questionnaire for the three input methods.

our experiment of the gestural keyboard, which improved the design of the SDK and made it less appealing to participants. In the previous experiment, instead, their participants had appreciated the idea of a keyboard that facilitates code entry by using shortcuts for programming constructs. The smaller screen size used in our experiment may also have contributed to obtain such a result.

The typing speeds obtained by the methods tested in our experiment are much lower than those obtained in common typing tasks (e.g., authors generally report average speeds of about 15–25 wpm with the baseline QWERTY layout [7], [8], [11]). Such a difference is due to the greater difficulty of entering code instead of simple text phrases. Additionally, the code used in our experiments is much longer than the short phrases used for ordinary text entry experiments ([22]).

As in previous tests [12] with similar gesture-based applications, in the trials with the gestural interface participants were able to look at the gesture set on a sheet of paper. This caused an unmeasured advantage for the gestural interface. In real editing situations such help would not be visible. A help function could be included in the user interface, but consulting it would be slow.

A limitation of the present study is its brevity. We only tested the first impact of our keyboard on users. Although this is of great importance, it says nothing about the evolution of its usability over time. A longitudinal study will be necessary to know the shape of the learning curve for being expert with the keyboard and its gestures. Another limitation is the small number (15) of participants.

Improving the discoverability of the gestures associated to programming operations is a necessary step not included in the current study and is programmed for the future. We will also consider to compare the gestural keyboard to other keyboard layouts and optimizations, e.g. the use of autocompletion. Future work possibly include also the production of the keyboard for the Android market, the testing of the gestural technique in different situations of code editing and the application of the same idea (a gestural keyboard) to different domains.

REFERENCES

- [1] I. Almusaly and R. Metoyer, “A syntax-directed keyboard extension for writing source code on touchscreen devices,” in *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. IEEE, 2015, pp. 195–202.
- [2] F. Raab, C. Wolff, and F. Echtler, “Refactorpad: editing source code on touchscreens,” in *Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems*. ACM, 2013, pp. 223–228.
- [3] B. Biegel, J. Hoffmann, A. Lipinski, and S. Diehl, “U can touch this: touchifying an ide,” in *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*. ACM, 2014, pp. 8–15.
- [4] N. Tillmann, M. Moskal, J. De Halleux, M. Fahndrich, and S. Burkhardt, “Touchdevelop: app development on mobile devices,” in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 39.
- [5] S. Kim, J. Son, G. Lee, H. Kim, and W. Lee, “Tapboard: making a touch screen keyboard more touchable,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2013, pp. 553–562.
- [6] I. S. MacKenzie and S. X. Zhang, “The design and evaluation of a high-performance soft keyboard,” in *Proc. of CHI ’99*, 1999, pp. 25–31.
- [7] S. Zhai, M. Hunter, and B. A. Smith, “The metropolis keyboard - an exploration of quantitative techniques for virtual keyboard design,” in *Proc. of UIST ’00*. New York, NY, USA: ACM, 2000, pp. 119–128.
- [8] X. Bi, B. A. Smith, and S. Zhai, “Quasi-qwerty soft keyboard optimization,” in *Proceedings of CHI ’10*. ACM, 2010, pp. 283–286.
- [9] ———, “Multilingual touchscreen keyboard design and optimization,” *Human-Computer Interaction*, vol. 27, no. 4, pp. 352–382, 2012.
- [10] P.-O. Kristensson and S. Zhai, “Shark2: a large vocabulary shorthand writing system for pen-based computers,” in *Proc. of UIST ’04*. NY, USA: ACM, 2004, pp. 43–52.
- [11] V. Fuccella, M. De Rosa, and G. Costagliola, “Novice and expert performance of keyscratch: A gesture-based text entry method for touchscreens,” *IEEE Transactions on Human-Machine Systems*, vol. 44, no. 4, pp. 511–523, 2014.
- [12] V. Fuccella, P. Isokoski, and B. Martin, “Gestures and widgets: performance in text editing on multi-touch capable mobile devices,” in *Proceedings of CHI ’13*. ACM, 2013, pp. 2785–2794.
- [13] L. Findlater, B. Lee, and J. Wobbrock, “Beyond qwerty: augmenting touch screen keyboards with multi-touch gestures for non-alphanumeric input,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2012, pp. 2679–2682.
- [14] P. Ihantola, J. Helminen, and V. Karavirta, “How to study programming on mobile touch devices: interactive python code exercises,” in *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*. ACM, 2013, pp. 51–58.
- [15] V. Fuccella and G. Costagliola, “Unistroke gesture recognition through polyline approximation and alignment,” in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM, 2015, pp. 3351–3354.
- [16] Y. Li, “Protractor: a fast and accurate gesture recognizer,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2010, pp. 2169–2172.
- [17] J. O. Wobbrock, M. R. Morris, and A. D. Wilson, “User-defined gestures for surface computing,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2009, pp. 1083–1092.
- [18] C. Parnin, C. Görg, and S. Rugaber, “Codepad: interactive spaces for maintaining concentration in programming environments,” in *Proceedings of the 5th international symposium on Software visualization*. ACM, 2010, pp. 15–24.
- [19] M. Hesenius, C. D. O. Medina, and D. Herzberg, “Touching factor: software development on tablets,” in *International Conference on Software Composition*. Springer, 2012, pp. 148–161.
- [20] S. McDirmid, “Coding at the speed of touch,” in *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*. ACM, 2011, pp. 61–76.
- [21] I. Almusaly, R. Metoyer, and C. Jensen, “Syntax-directed keyboard extension: Evolution and evaluation,” in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct 2017, pp. 285–289.
- [22] I. S. MacKenzie and R. W. Soukoreff, “Phrase sets for evaluating text entry techniques,” in *Proc. of CHI EA ’03*. ACM, 2003, pp. 754–755.
- [23] I. S. MacKenzie, “A note on calculating text entry speed,” last visit 2014, <http://www.yorku.ca/mack/RN-TextEntrySpeed.html>.
- [24] R. W. Soukoreff and I. S. MacKenzie, “Metrics for text entry research: an evaluation of msd and kspc, and a new unified error metric,” in *Proc. of CHI ’03*. New York, NY, USA: ACM, 2003, pp. 113–120.
- [25] I. S. MacKenzie and K. Tanaka-Ishii, *Text entry systems: Mobility, accessibility, universality*. Morgan Kaufmann, 2010.

Evaluation of A Visual Programming Keyboard on Touchscreen Devices

Islam Almusaly

Oregon State University

almusali@oregonstate.edu

Ronald Metoyer

University of Notre Dame

rmetoyer@nd.edu

Carlos Jensen

Oregon State University

carlos.jensen@oregonstate.edu

move	turn	jump	draw	print	
Number	Get variable	repeat times	if	Create Function	Color RGB
Arithmetic	Set variable	count with	Boolean	Text	Compare
Math Keys	Variable Keys	Loop Keys	Logic Keys	Function Keys	Color Keys
Text Keys	List Keys	repeat while	↑	↓	↔

Fig. 1: A soft keyboard for inputting blocks.

Abstract—Block-based programming languages are used by millions of people around the world. Blockly is a popular JavaScript library for creating visual block programming editors. To input a block, users employ a drag-and-drop input style. However, there are some limitations to this input style. We introduce a custom soft keyboard to input Blockly programs. This keyboard allows inputting, changing or editing blocks with a single touch. We evaluated the keyboard users' speed, number of touches, and errors while inputting a Blockly program and compared its performance with the drag-and-drop method. Our keyboard reduces the input errors by 68.37% and the keystrokes by 47.97%. Moreover, it increases the input speed by 71.26% when compared to the drag-and-drop. The keyboard users perceived it to be physically less demanding with less effort than the drag-and-drop method. Moreover, participants rated the drag-and-drop method to have a higher frustration level. The Blockly keyboard was the preferred input method.

I. INTRODUCTION

Computer science jobs are increasing with more than 50% of all science, technology, engineering, and math (STEM) jobs projected to be in computer science-related fields by the year 2018 [1]. In addition, the Computer Science for All (CS for All) initiative is aimed at enabling all American students at K-12 schools to learn computer science. One of the common ways to introduce computer science is block-based programming. Alice, Scratch, and Blockly are examples of such block-based languages [2], [3], [4]. They were chosen because they offer some advantages over textual languages for novice programmers.

One of the advantages that blocks programming environments offer is that they eliminate syntax issues because they represent program syntax trees as compositions of visual blocks. For this reason, they are being used by millions of people of all ages and backgrounds and they offer many other

U.S. Government work not protected by U.S. copyright

advantages to the novice programmer. Despite their advantages, they have their drawbacks. One of these drawbacks is the time and the number of blocks it takes to compose a program in the block-based interface compared to the text-based alternative [5]. Dragging blocks from a toolbox is slower than typing.

Some attempts have been made to blur the line between blocks and text programming [6], [7], [8], [9]. Few of them employed a strategy that allowed blocks to be typed using the keyboard. However, these researches did not focus on blocks input performance. Instead, they sought to ease the transition from blocks to text-based programming while we attempt to ease the input of existing block-based languages, especially on touchscreen devices.

Most block-based programming environments rely on the mouse as the primary means of inputting blocks. Using touchscreen devices, blocks can only be input by drag-and-drop. However, using the drag-and-drop input method has its disadvantages when compared to the point-and-click input method. The point-and-click input method is faster, more accurate, and it is preferred over the drag-and-drop for adults and children alike [10], [11]. Drag-and-drop requires careful manipulation of blocks to insert them in the correct place. The careful manipulation requirement adds physical and cognitive demands. These demands affect users negatively, especially people with motor disabilities or in the case of children. The drag-and-drop interaction also changes as the canvas is zoomed in or out. A zoomed-out canvas makes connecting blocks more difficult as the blocks' connectors become smaller, making it more challenging to aim for. This drag-and-drop entry method also does not take advantage of all fingers for inputting the blocks. In addition, blocks have many options that can be changed. Unfortunately, these options must be changed by manipulating small icons. These factors make block entry a slow and difficult process. Furthermore, these factors might lead to frustrated users, which might affect the performance and adoption of visual languages. For these reasons, we are interested in addressing these issues.

We created a custom soft keyboard to input blocks on touchscreen devices. This keyboard was made specifically as a drag-and-drop alternative and enables programmers to input blocks using a point-and-click interaction style. We seek to reduce the time required and number of errors when inputting programs using the keyboard. Moreover, a faster and

more accurate way to input blocks could reduce physical and temporal demands thus reducing frustrations. Finally, a faster, more accurate, and more efficient input method will make blocks entry more appealing and accessible to a wider range of users.

In summary, then, this paper contributes the following:

- A first attempt at identifying three block specific input metrics; for measuring speed, efficiency, and accuracy
- A first attempt at quantifying blocks' input performance on touchscreen devices using the three different measures
- A soft keyboard design for blocks input
- A user study of the keyboard's input performance compared to the drag-and-drop
- Reflections on research opportunities for blocks input

II. RELATED WORK

A. Blocks-based Programming

Blocks-based languages are used to gradually introduce novices to programming. In these languages, programs are constructed by connecting blocks via connectors. Alice, Scratch, and Blockly are examples of such block-based languages and environments [2], [3], [4]. They are engaging millions of children with programming through drag-and-drop [12]. Moreover, they were designed explicitly with learners in mind [13] and thus many novice students prefer block-based over the text-based languages [5]. These visual programming languages and environments help students overcome common barriers that novice programmers encounter such as selection, coordination, and use barriers [14]. Block-based languages help novice programmers overcome these barriers by using recognition of blocks instead of recall of syntax and preventing a block from being connected to the wrong connector.

Our keyboard, however, is different because it is targeted to both novice and experts users. It supports both use types with better input efficiency by reducing the number of touches required to input blocks, leading to faster speeds and reduced errors. Novice users might benefit from the reduced physical demand while experts might benefit from the increased speed. Novice and experts will benefit from reduced errors.

B. Input Methods

Block-based languages rely heavily on mouse and keyboard input. When it comes to touchscreen devices, the input styles vary. Touch, voice, and gesture controls are examples of such input styles. A voice-driven tool to input blocks was introduced by Wagner [15]. Using this tool, children with motor disabilities can input block hands free using their voice. The standard approach to input blocks is by using drag-and-drop with the on screen QWERTY keyboard. This soft keyboard is the default keyboard on most of these devices. The QWERTY keyboard, however, is not suited for every input scenario.

Keyboards are designed to minimize discomfort, speed input, or both for a specific task. Most keyboards are designed and evaluated for text entry [16]. However, the design of the blocks keyboard is different because it was designed

and evaluated with blocks entry. The main inspiration of our keyboard is the syntax-directed-keyboard extension which was designed with program input in mind [17]. Instead of using the Java syntax to input, the blocks keyboard uses Blockly's rules. However, designing a keyboard for blocks input is different than textual programming language like Java. For instance, blocks are input via dragging which is vastly different from keyboards usage. There are few attempts on easing programming and collaboration on tablets. However, the effects of the drag-and-drop mechanism on blocks input have not fully explored. In this paper, we will evaluate the current block-based input method and compare it against our keyboard. We discuss in detail the design of the keyboard in the following section.

III. THE BLOCKS KEYBOARD DESIGN

The following sections explain the design motivations and how the keyboard works. The keyboard also went through several design iterations. We describe the main iterations that led to the keyboard's final version in this section.

A. How Does It Work

The keyboard works like text entry keyboards, however, instead of inputting letters it inputs blocks. Each key inputs a block or changes a field. Figure 3 shows the current layout of the keyboard. Once a block is inserted, the keyboard selects and highlights the first unoccupied input connector. The highlighted connector acts as a cursor. Navigating the blocks through their connectors is available by using the arrow keys 2. When a connector is selected, it gets highlighted. The keyboard's keys will be enabled or disabled according to the highlighted connector (cursor). The "if" block, for example, does not allow numbers to be connected to its "condition" connector. Thus, the "Number" and "Arithmetic" keys will be disabled if the condition connector is selected. The grayed-out keys in Figure 3 are disabled. The top row will list the options for the current block, the block with a highlighted connector. As a user moves through the blocks, the top row updates the list of options automatically. For example, the right side of Figure 4 shows how the top row displays the options for the "Compare" block.



Fig. 2: Examples of highlighted connectors (different cursor locations).

B. Block Frequency

There are many ways to organize the keys in the keyboard. Just like many keyboard designs that utilize the letter or word frequencies to lay out the keys, we utilize the blocks' frequencies. To do this, we identified Blockly program sources in Code Studio, a website offering online courses and used by

millions of students [12]. It relies heavily on Blockly to teach programming concepts. Therefore, we chose it as a reference for Blockly programs. We counted the frequency of each block that was asked to be input in all the offered activities. However, if the activity asks the users to input less than 10 blocks we did not include that activity in the statistics. We chose not to include such activities because they have fewer blocks that do not represent a common block-base program. Commonly, an activity with few blocks just teaches how to input blocks rather than teaching programming concepts or problem solving. This left us with 47 Blockly programs from courses two, three, and four. We found that the average input task consists of 19 blocks. Table I shows the frequency of block types. The Code Studio activities did not ask the students to input all the block types in Blockly as can be observed in the table.

TABLE I: The frequency of inputted blocks.

Block Type	Frequency
function call	34.1%
number	23.7%
get variable	14.4%
repeat time	9.2%
arithmetic	5.5%
set variable	5.4%
for loop	3.2%
if	1.6%
function define	1.3%
color with	0.6%
repeat while	0.1%

The keyboard underwent many iterations and the blocks' statistics served as a guide for placing the keys throughout these iterations. First, we placed block types as keys from most frequent (right of the top row) to least frequent (left of the bottom row). Then, we placed block categories that contain a list of blocks with similar functionality, as keys from most frequent to least frequent after the block keys based on the sum of their block usage. By this stage, we had the first version of the keyboard which is shown in Figure 3. However, the block types were scattered across the keyboard. We grouped the keys by swapping the next frequent key of the same category for each key with the key underneath it. For example, "Arithmetic" is the next frequent key after the "Number" key from the same category. We swapped "Arithmetic" with "if". After repeating the same process for the rest of the keys we had the a newer version. For the newer version, we manually moved the "repeat while" key underneath its category because it is the least frequent block type to maintain the grouping. After placing the blocks and their categories, three keys were not assigned. We choose to assign them to "Text", "Boolean", and "Compare" blocks to enable access for other data type blocks and the comparison block. Finally, we listed the predefined functions in the first-row because function call blocks have the highest frequency of all. The first-row acts like a dynamic placeholder for blocks and their options. It lists the options of the last inputted block alongside the predefined functions. For instance, when the "Arithmetic" block is inputted, the dynamic row lists all of its five options: +, -, ×, ÷, and \wedge .

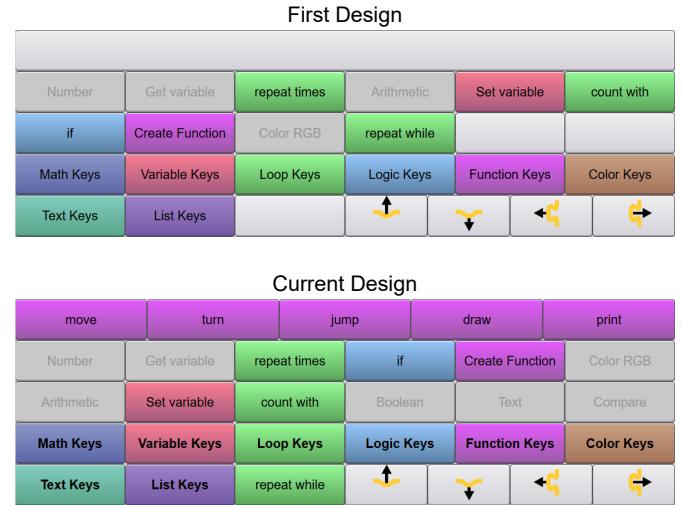


Fig. 3: The first and current versions of the keyboard design.

C. User-Interface Design Principles

While the keyboard was evolving, we adhered to the general user-interface design principles listed by Wickens et al [18]. These general design principles help to ensure the keyboard's ease of use and adaptability.

1) *Make invisible things visible:* Opposite to the drag-and-drop, the Blockly keyboard lists the block's options making them readily available with one touch. These options are hidden in a drop-down menu in the drag-and-drop interface. Figure 4 shows how the keyboard displays the options for the "Compare" block as opposed to the hidden options of different blocks.

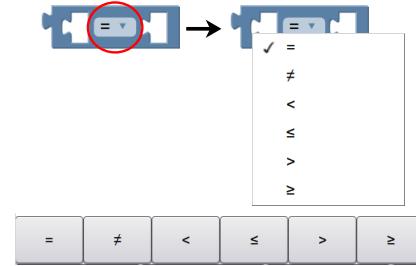


Fig. 4: An example of how the keyboard makes the options more accessible.

2) *Consistency and standards:* Each block and its options are expressed as a key with the same height and width except the dynamic top row. The size of a block, yet, changes and its connectors move. In addition, keys have two actions, which are input a block or change an option. This is consistent throughout the keyboard. Furthermore, keys with the same functionality are grouped together.

3) *Error prevention, recognition, and recovery:* Using the drag-and-drop method in Blockly, users can connect the wrong blocks together. Blockly then disconnects the wrong blocks without prompting the user, which might be confusing. To prevent errors from occurring in the first place, our keyboard

disables keys to prevent inputting the wrong block in the wrong place.

4) Memory: The keyboard keys are named and colored which utilize see-and-point instead of remember-and-type. Our keyboard exposes the most common blocks instead of hiding them inside a toolbox and reveals their options. The users do not have to remember the locations of the common block inside the toolbox nor do they need to search for the options from their menu. This reduces the reliance on memory.

5) Flexibility and efficiency of use: The keys and the option keys act as shortcuts and accelerators. Our keyboard gives the users the option to speed up frequent actions by listing the most frequent inputted blocks, the "Function call" blocks. In addition, the dynamic top row gives shortcuts for the block's options.

6) Simplicity and aesthetic integrity: The keyboard's keys are aligned with uniform width and height to make them aesthetically pleasing with a simple design. To make information appear in a natural order, the options and keys are presented from left to right based on their usage frequency.

IV. USER STUDY

The formal user study was designed to compare the input performance of the keyboard to the drag-and-drop method when inputting a Blockly program. We presented the participants with a letter sized paper that has a Blockly program printed in colors. The participants then were asked to copy the program with both input methods. We chose a copying task as opposed to a programming task to avoid the confounding factors of the cognitive aspects of programming. When the keyboard is shown, the drag-and-drop is disabled to measure the native keyboard performance. We used an iPad 4 for our user study. Both input methods are implemented in JavaScript. The same JavaScript code was used to measure and log the participants' interactions, time, and errors. The error types presented earlier are collected by the instrumented JavaScript code.

A. Participants

The study participants consisted of 14 male and two female students. All participants volunteered for the study in response to an email message circulated to the students in the computer science department at Oregon State University. Two participants were graduate students and 14 were undergraduate students. All participants had never used Blockly. Nine of the participants were not familiar with block-based programming. None of the participants used a tablet device to input any block-based program. Ten participants reported having a tablet device. The participants were compensated for participating in the study.

B. Apparatus

The input task was performed using an iPad (4th generation) with a 9.7-inch 2048x1536 (264 ppi) multi-touch display. Both input methods were running on the Safari web browser under iOS 10.3.3. The input methods use JavaScript to handle touch events.

There are many measures to evaluate an input method. We used three common input measures for evaluating user performance with the keyboard, namely, the input accuracy, efficiency, and speed.

1) Accuracy: Keyboards affect accuracy in various ways. The more prone a keyboard is to errors the less accurate it is. Errors include misspellings or typos when using a text entry keyboard. However, the inputted elements in visual programming are blocks. Thus, error types will be different. We defined errors to be the actions that the user did not intend to do when inputting blocks. These actions are misplacing a block, failing to connect a block to another, selecting the wrong field, and inputting the wrong block. Misplacing a block occurs when the participant inputs a block and connects it to the wrong block. When the participant inputs a block far from another block's connector, the inserted block will not be connected. This error is counted as failing to connect a block to another. Selecting the wrong option occurs when a participant did not choose a block's field correctly. When a participant inputs the wrong block, it is counted as inputting the wrong block. The error rate is the sum of all the error types.

2) Efficiency: Keystrokes per character (KSPC) is frequently used characteristic of text entry methods [19]. For a given text entry method, it measures the number of keystrokes required, on average, to generate a character of text. However, blocks are not characters. We defined a similar block-based characteristic for block entry methods. The keystrokes per block (KSPB) is a measurement of the number of keystrokes required, on average, to generate a block. Thus, a block entry technique with lower KSPB is more efficient.

3) Speed: To evaluate the speed of text entry techniques, words per minute (WPM) is used. WPM, as the name implies, is the average number of words that can be inputted in a minute by a text entry method, assuming five letters per word. We defined a related measure for evaluating the speed of block entry methods. Blocks per Minute (BPM), equation 1, is the average number of blocks that can be inputted in one minute by a block entry technique. A faster entry method is the one with a higher BPM.

$$BPM = \frac{Number\text{ of }Blocks}{Time} \quad (1)$$

C. Study Design

To study the difference between the two input methods, we used a within-subjects design with repeated measures. The independent variable was the input method used to complete the task and the study consisted of two treatments: the drag-and-drop, and the keyboard. We asked each participant to enter the Blockly program using each input method. We counterbalanced the order of the treatments by dividing the subjects into two groups. One group started with the drag-and-drop, followed by the keyboard, and the other started with the keyboard followed by the drag-and-drop. The dependent variables were the time, errors, and the number of touches. These variables are used to calculate the speed (BPM), accuracy (%), and

efficiency (KSPB) that were mentioned earlier. We measured these variables for each input method, independently.

D. Procedure

To see how the input methods are going to perform in a common block-base program, we collected statistics from Code.org website. We asked the participants to input the one Blockly program, which consisted of 29 blocks. In addition, we designed the task to conform to the collected statistics from Table I. The input task consists of 10 “function call” blocks, seven “number” blocks, four “get variable” blocks, three “repeat” blocks, two “arithmetic” blocks, two “set variable” blocks, and one “for” block. Figure 5 shows the input task.

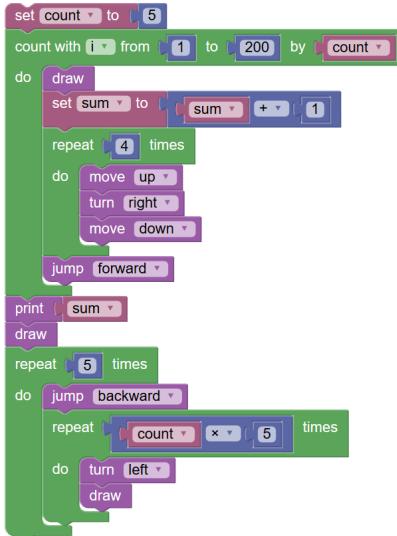


Fig. 5: The Blockly input task.

We ran the study in a lab setting one participant at a time. After signing an informed consent document, each participant was randomly assigned to one of the two experimental conditions as described above. Each participant was given a tutorial on how to use the drag-and-drop and the keyboard to input a Blockly program. We encouraged the participants to ask any questions that they might have during the study. The participants then carried out the input task using the two treatments. After each task, we asked the participants to complete a NASA Task Load Index (NASA-TLX) questionnaire for assessing subjective mental workload [20]. When the task had been completed, we asked participants to complete a post-session questionnaire about their experience. The average duration of a session with each participant was 30 minutes.

V. RESULTS

Our initial hypothesis was that users would input a Blockly program faster, more efficiently, and with fewer errors when using the keyboard as compared to the drag-and-drop. Thus, our null hypothesis for all analyses is that there is no significant difference between the distributions of corresponding performance measures across the two input methods. For all

measurements, we used a paired t-test analysis. Figure 6 summarizes the performance of each input method.

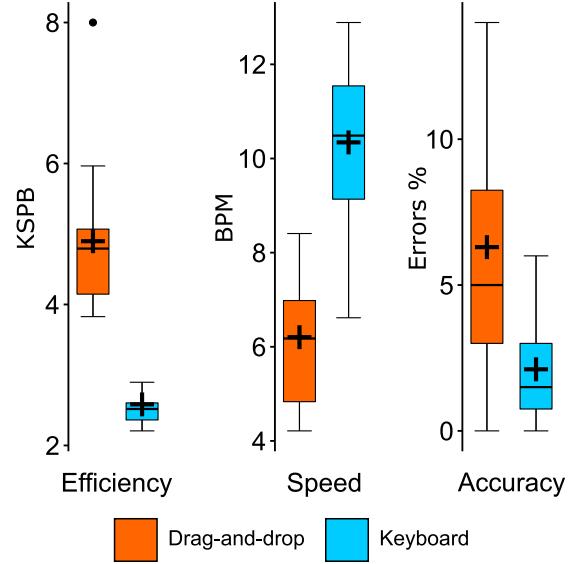


Fig. 6: The efficiency, speed, and accuracy for both the drag-and-drop and keyboard. The mean is shown with the “+” sign.

A. Accuracy

Participants’ mean errors for the drag-and-drop and the keyboard methods were 6.13% (SD: 4.21%) and 1.93% (SD: 1.84%), respectively. This represents a 68.37% reduction in errors with the keyboard. There was a convincing statistical evidence for an effect of the input method on errors ($t_{(15)} = 3.5564, p < .01$). See the third column in Figure 6.

B. Efficiency

The average touches to input the same Blockly program with the keyboard were fewer, 72.81 touches (SD: 6.19), than the drag-and-drop, 139.94 touches (SD 30.27). This means that the drag-and-drop has a 4.83 KSPB compared to the 2.51 KSPB for the keyboard. This represents a decrease of 47.97% in the keystrokes required to input a block. There is a convincing statistical evidence for an effect of the keyboard on the number of touches ($t_{(15)} = 9.0083, p < .01$). The first column in Figure 6 shows the difference in the number of touches between the two input methods for the exact program.

C. Speed

Participants, on average, took longer 174.88 seconds (SD: 32.74) and 299.31 seconds (SD: 68.18) to input the program on the keyboard and drag-and-drop, respectively. In another word, the keyboard has a speed of 9.95 BPM while the drag-and-drop has a speed of 5.81 BPM. The keyboard is 71.26% faster than the drag-and-drop method and the pair-wise t-test shows a significant difference in the speed between them ($t_{(15)} = 7.9963, p < .01$). The second column in Figure 6 gives us a picture of the performance with respect to time.

D. NASA-TLX

Table II shows the mean response values for the RAW NASA-TLX measures. While there was no statistical evidence for the mental demand, temporal demand, or performance, there was convincing statistical evidence for an effect of input method on the other measures. These are the physical demand ($t_{(15)} = 4.332, p < .01$), effort ($t_{(15)} = 2.9929, p < .01$), and frustration level ($t_{(15)} = 3.7284, p < .01$). Figure 7 summarizes the TLX questionnaire results.

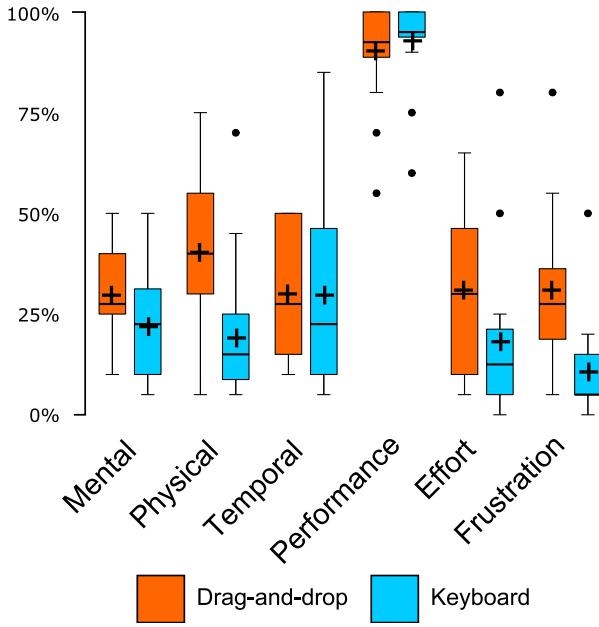


Fig. 7: A summary of the NASA-TLX measures.

TABLE II: NASA-TLX measures comparison (mean responses) between the drag-and-drop and the keyboard. The percentage column shows the decrease rate of the keyboard. A negative value indicates an increase.

TLX Measure	Dragging	Keyboard	Percentage
Mental Demand	29.69	21.88	26.32%
Physical Demand	40.31	19.06	52.71%
Temporal Demand	30.00	29.68	1.04%
Performance	90.31	92.81	-2.77%
Effort	28.13	18.12	41.41%
Frustration	30.94	10.63	65.66%

E. Participants' Preference

After inputting the task with both input methods, participants completed a questionnaire about their overall experience with the keyboard. 75% of the participants indicated that they are likely to use the keyboard and 19% of them felt neutral. Only one participant indicated that he/she is not likely to use the keyboard in the future. In addition, 94% of participants found the keyboard to be helpful for inputting the Blockly program. 88% of the participants thought it was easy to adapt to the keyboard and 94% thought it was easy to use. 75% of the participants thought that the design of the keyboard is good and the rest felt neutral about the design. All the participants

felt that they were efficient when using the keyboard. Figure 8 shows each question and a box-plot of the participants' average answers to each question.

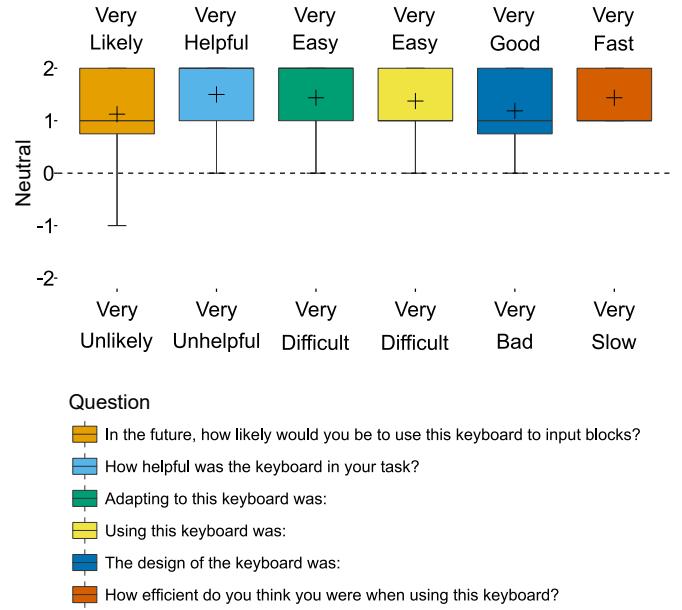


Fig. 8: The results of the post-study questionnaire. Each column represents a question. The boxplots show the average response on a 5-point scale.

VI. DISCUSSION

The results of the study show that users performed blocks input better when using the keyboard as measured by BPM, KSPB, and errors. It is worth noting that these results were obtained after only 10 minutes of practice. It is expected to see a better input performance from a point-and-click input style like our keyboard when compared to a drag-and-drop method as was mentioned earlier. One explanation for this result is the shorter distance that fingers must travel when using the keyboard. Moreover, the average key size of the keyboard is larger than the average drag-and-drop touch targets. Per Fitts' law, the shorter distance and the larger target size will positively affect the speed of the input task [21]. This can be seen in Figure 9. In this Figure, the touch locations are spread across a larger area for the drag-and-drop, by contrast, they are restricted to a smaller area for the keyboard.

A. Accuracy

The keyboard allows the users to input blocks with 68.37% fewer errors than the drag-and drop. There are many reasons for this result. First, the keyboard inputs each block to the highlighted connector automatically. Consequently, the errors from connecting a block to the wrong connector are eliminated. Second, the errors are reduced because of the large size of the keys compared to the toolbox or the block's options menu sizes. Finally, as can be seen in Figure 9, that the participants' fingers travel longer distances while dragging blocks. The keyboard, nevertheless, requires no dragging.

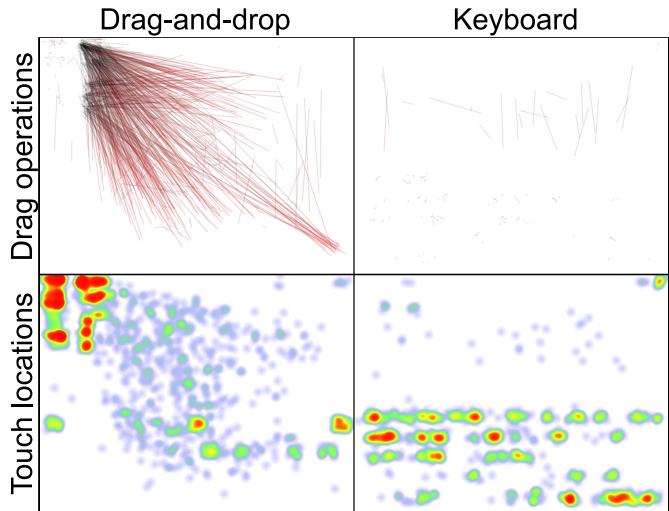


Fig. 9: A visualization of the dragging and touching locations when inputting the Blockly program for all participants. The drag operation lines start from black and end in red.

Despite that, few participants tried to drag the blocks because they thought that they could drag blocks when using the keyboard even though they were told otherwise. The same Figure shows that the touch locations for the keyboard are more confined whereas they are more scattered for the drag-and-drop method. Therefore, the chance of introducing errors is increased with more scattered touches and longer travel distances. These reasons combined make the keyboard a more accurate way to input blocks.

B. Efficiency

There is a considerable difference between our keyboard and the drag-and-drop method when it comes to the number of touches. Our keyboard allows inputting blocks with almost half the number of touches without dragging (47.97% fewer touches). This reduction happens largely because of two reasons. First, the reduction in errors means less need for corrections. Thus, reducing the number of touches. The second reason is the ability to change the options without touching the drop-down menus to open them. This requires one less touch each time an option needs to be changed. Many blocks rely on changing options and this impacts the efficiency of the drag-and-drop method negatively.

C. Speed

The keyboard is exceedingly fast in comparison to drag-and-drop (71.26%). The reduced errors and keystrokes lead to this boost in the input speed. In addition, the automatic insertion of the blocks in the right connector without the need to drag is another area that helped the keyboard's speed. The blocks, however, need careful and precise positioning when dragging which slows the input.

Although the keyboard is fast, we suspect that the keyboard will be even faster after a longer period of use. Just like all keyboards, the key locations will be memorized and the visual

scanning will take less time resulting in faster input speed. The input speed result in our study is for novice users. An expert user of a keyboard will input with higher speed [16]. However, the same thing cannot be said for the drag-and-drop method. The blocks reshape themselves after being connected to other blocks or after changing the options. For example, renaming the variables or changing the number in blocks will change the size of the block. Figure 5 shows how blocks with the same type have different shapes and connector locations, making dragging operations too difficult to memorize. In Figure 9, we can see how one program has many ways of dragging operations. Therefore, we suspect that the keyboard will be much faster after practice than the drag-and-drop.

D. NASA-TLX and Participants' Preference

Preferring a point-and-click style like the keyboard over the drag-and-drop method was shown by different studies [10], [11]. Our keyboard is no different. The NASA-TLX and the participants' feedback demonstrate that participants prefer the keyboard over the drag-and-drop method. Inputting blocks with fewer touches makes the keyboard less physically demanding which was confirmed by our participants' perceived physical demand (53% less physical demand). This may also affect the perceived effort (36% less effort). However, the lower frustration may be caused by the lower errors and the faster input (66% less frustration). From the post-task questionnaire, the majority of the participants preferred to use the keyboard and found it to be easy to adapt and use. The participants' preference is clear from their comments too. For example, participant one said, "Keyboard was more easier than drag and drop". While participant eight said, "The automatic movement of the cursor was better than the drag and drop function it not only reduces the work of properly pairing two parts together but also was easy and smart". Participant nine said, "The keyboard helps to reduce the dragging time which is helpful". Participants 10 and 11 respectively said, "I prefer the keyboard. It was much faster than moving the blocks around" and "adapting to the keyboard was so easy and natural and faster than the drag and drop method".

E. Limitations

Just like other keyboards, there are some limitations for our keyboard. These results were achieved for a specific task. Any Blockly program that does adhere to the collected statistics will perform differently. In that case, however, we can safely assume that the keyboard input performance will not suffer dramatically. We assume that because the low number of touches, the faster speed, and the reduced errors will still hold due to the lack of dragging and the confined keyboard area when inputting different Blockly programs. We tested our keyboard on the original Blockly code. However, there are many derivatives of Blockly. Each one of them will have different input performance. Our keyboard holds a list of commands and their key names. One can change this list to call different or new blocks to accommodate different visual languages if they run on JavaScript.

VII. CONCLUSIONS AND FUTURE WORK

We presented a keyboard alternative to drag-and-drop for inputting Blockly programs. We discussed the motivation and the design of this keyboard. The user study showed how our keyboard surpasses the drag-and-drop method in terms of accuracy, efficiency, and speed when inputting a Blockly program. In addition, most of the participants preferred the keyboard and found it to be easy to use and learn. They also perceived it to have less physical demand, less effort, and less frustration level.

This keyboard opens the door to potential future work. The results were obtained after a 10-minute practice. Better results are expected after prolonged use. A longitudinal study of the keyboard will show how far the input performance will go. The keyboard might make blocks input accessible for people with visual impairments because many of them rely on keyboards [22]. It could be beneficial for people with motor skill disabilities to input blocks without dragging. Another area of interest would be to see how a custom keyboard like this performs in other visual programming languages. Although our keyboard was tested on adults, children from different age groups may benefit from using such a keyboard differently because of the variation of their abilities. This makes children potential participants for future work. The keyboard could serve as an intermediate step to learning how to write textual programs because Blockly has a mapping of blocks to JavaScript, Python, PHP, Lua, and Dart. A study of the potential impact of this keyboard as a transitional step toward text-based languages could also prove fruitful. Lastly, enabling the two input methods at the same time might bring the positives from both. We are now extending the keyboard capabilities to work in conjunction with the drag-and-drop method.

ACKNOWLEDGMENT

We would like to thank the participants in our study and those who have given extensive comments on earlier versions of this paper.

REFERENCES

- [1] A. P. Carnevale, N. Smith, and M. Melton, "Stem: Science technology engineering mathematics." *Georgetown University Center on Education and the Workforce*, 2011.
- [2] S. Cooper, W. Dann, and R. Pausch, "Alice: a 3-d tool for introductory programming concepts," in *Journal of Computing Sciences in Colleges*, vol. 15, no. 5. Consortium for Computing Sciences in Colleges, 2000, pp. 107–116.
- [3] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman *et al.*, "Scratch: programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.
- [4] N. Fraser *et al.*, "Blockly: A visual programming editor," *URL: https://code.google.com/p/blockly*, 2013.
- [5] D. Weintrop and U. Wilensky, "To block or not to block, that is the question: students' perceptions of blocks-based programming," in *Proceedings of the 14th International Conference on Interaction Design and Children*. ACM, 2015, pp. 199–208.
- [6] J. Monig, Y. Ohshima, and J. Maloney, "Blocks at your fingertips: Blurring the line between blocks and text in gp," in *Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE*. IEEE, 2015, pp. 51–53.
- [7] D. Bau, D. A. Bau, M. Dawson, and C. Pickens, "Pencil code: block code for a text world," in *Proceedings of the 14th International Conference on Interaction Design and Children*. ACM, 2015, pp. 445–448.
- [8] D. Bau, "Droplet, a blocks-based editor for text code," *Journal of Computing Sciences in Colleges*, vol. 30, no. 6, pp. 138–144, 2015.
- [9] M. Kölling, N. C. Brown, and A. Altadmri, "Frame-based editing: Easing the transition from blocks to text-based programming," in *Proceedings of the Workshop in Primary and Secondary Computing Education*. ACM, 2015, pp. 29–38.
- [10] I. S. MacKenzie, A. Sellen, and W. A. Buxton, "A comparison of input devices in element pointing and dragging tasks," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 1991, pp. 161–166.
- [11] K. M. Inkpen, "Drag-and-drop versus point-and-click mouse interaction styles for children," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 8, no. 1, pp. 1–33, 2001.
- [12] Code.org, "Learn on code studio," 2017. [Online]. Available: <https://studio.code.org/>
- [13] D. Weintrop and U. Wilensky, "Bringing blocks-based programming into high school computer science classrooms," in *Annual Meeting of the American Educational Research Association (AERA)*. Washington DC, USA, 2016.
- [14] A. J. Ko, B. A. Myers, and H. H. Aung, "Six learning barriers in end-user programming systems," in *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*. IEEE, 2004, pp. 199–206.
- [15] A. Wagner, R. Rudraraju, S. Datla, A. Banerjee, M. Sudame, and J. Gray, "Programming by voice: A hands-free approach for motorically challenged children," in *CHI'12 Extended Abstracts on Human Factors in Computing Systems*. ACM, 2012, pp. 2087–2092.
- [16] I. S. MacKenzie, S. X. Zhang, and R. W. Soukoreff, "Text entry using soft keyboards," *Behaviour & information technology*, vol. 18, no. 4, pp. 235–244, 1999.
- [17] I. Almusaly and R. Metoyer, "A syntax-directed keyboard extension for writing source code on touchscreen devices," in *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. IEEE, 2015, pp. 195–202.
- [18] C. D. Wickens, S. E. Gordon, Y. Liu, and J. Lee, "An introduction to human factors engineering," 1998.
- [19] I. S. MacKenzie, "Kspc (keystrokes per character) as a characteristic of text entry techniques," in *International Conference on Mobile Human-Computer Interaction*. Springer, 2002, pp. 195–210.
- [20] S. G. Hart and L. E. Staveland, "Development of nasa-tlx (task load index): Results of empirical and theoretical research," *Advances in psychology*, vol. 52, pp. 139–183, 1988.
- [21] P. M. Fitts, "The information capacity of the human motor system in controlling the amplitude of movement." *Journal of experimental psychology*, vol. 47, no. 6, p. 381, 1954.
- [22] S. Ludi, "Position paper: Towards making block-based programming accessible for blind users," in *Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE*. IEEE, 2015, pp. 67–69.

CodeDeviant: Helping Programmers Detect Edits That Accidentally Alter Program Behavior

Austin Z. Henley

Department of Electrical Engineering & Computer Science
University of Tennessee
Knoxville, Tennessee 37996-2250
azh@utk.edu

Scott D. Fleming

Department of Computer Science
University of Memphis
Memphis, Tennessee 38152-3240
Scott.Fleming@memphis.edu

Abstract—In this paper, we present CodeDeviant, a novel tool for visual dataflow programming environments that assists programmers by helping them ensure that their code-restructuring changes did not accidentally alter the behavior of the application. CodeDeviant aims to integrate seamlessly into a programmer’s workflow, requiring little or no additional effort or planning. Key features of CodeDeviant include transparently recording program execution data, enabling programmers to efficiently compare program outputs, and allowing only apt comparisons between executions. We report a formative qualitative-shadowing study of LabVIEW programmers, which motivated CodeDeviant’s design, revealing that the programmers had considerable difficulty determining whether code changes they made resulted in unintended program behavior. To evaluate CodeDeviant, we implemented a prototype CodeDeviant extension for LabVIEW and used it to conduct a laboratory user study. Key results included that programmers using CodeDeviant discovered behavior-altering changes more accurately and in less time than programmers using standard LabVIEW.

I. INTRODUCTION

A particularly difficult activity for programmers is understanding how their changes to code affect other parts of the program. Because software is made up of many inter-related code modules, a small change in one module can have cascading effects throughout the rest of the program. Moreover, code is often missing explicit information about the relationships between code modules (known as *hidden dependencies* [15]). To understand the full impact of a code change, programmers must possess a correct mental model of the source code. In fact, researchers have generally found that people require a rich mental model before they can organize information on an even less complex task, such as choosing the best camera to purchase [20]. However, forming such mental models about programs is a notoriously error-prone and time-consuming task due to the sheer size and complexity of modern software [33]. This is further complicated by the fact that programmers’ information needs are rapidly changing as they work through programming tasks [34], [35].

In this paper, we focus on a large class of code changes, known as *refactorings*, specifically in the context of visual programming environments. Refactoring aims to improve the design of a program by changing the structure of its code without altering its behavior [12], [31]. It has become a ubiquitous practice in software development [19], [28]. Surveys of pro-

grammers have indicated that programmers find refactoring to be an important part of the development process [7], and they believe it provides a variety of benefits, including improving readability and extensibility [19]. Studies of both textual and visual programming languages have provided some support for these views, empirically demonstrating the benefits refactoring can provide. In particular, studies of textual languages have shown that refactoring improved maintainability [21] and reusability [27] of code. Although little work has been done to study refactoring in visual languages, one study did find that programmers preferred code that had been refactored to remove code smells [43].

Despite refactoring’s popularity and benefits, it is often difficult for programmers, both of textual and visual languages, to perform. Most programming environments for textual languages provide features for automated refactorings, but programmers rarely use them [13], [19], [28], [30], [44]. Several reasons have been cited for the underutilization of such features. The tools are not trusted by programmers [44]; they provide unhelpful error messages [28]; and they have even been found to introduce bugs [4], [10], [39], [41], [44], [46]. However, refactoring manually is a tedious process that has also been found to be error prone in textual languages [13] as well as in visual languages [17].

A particularly difficult aspect of refactoring observed among textual-language programmers is ensuring that code changes did not alter the behavior of the program. Best practices suggest the use of test suites, which allow programmers to define correct program behavior and to be alerted whenever a code change causes a test to fail. Creating such software tests is a common approach to ensuring software quality in contemporary software development. However, refactorings may break the code that tests the software [24], [36], [39]. Moreover, software tests have also been found to be inadequate at finding refactoring errors [37], and having tests available during refactoring did not improve the quality of the refactorings produced by programmers [45]. To address these shortcomings of software testing, researchers have recently proposed tools to detect manual refactorings, automatically complete them, and validate their correctness (e.g., BeneFactor [13], GhostFactor [14], and WitchDoctor [11]). However, these tools do not support all types of refactorings.

To better understand the challenges of refactoring in visual languages, we conducted formative investigations of programmers of the visual dataflow language, LabVIEW, engaged in refactoring, and found that, similar to textual-language programmers, they also have considerable difficulties in attempting to validate their code changes. In particular, the programmers reported that they often introduce bugs unintentionally while refactoring. To cope with this issue, these programmers followed tedious strategies to detect behavior-altering changes, such as writing down program output on a piece of paper prior to the code change. An analysis of the programmers' refactorings found that not only were buggy refactorings common, but that they did indeed alter the program output unintentionally.

To address these issues of detecting behavior-altering changes, we propose CodeDeviant, a novel tool concept for visual programming language environments. The CodeDeviant tool compares the program output with a previous execution to determine whether the behavior has been altered. CodeDeviant aims to integrate seamlessly into programmers' workflows by not requiring any upfront effort (unlike creating software tests). Key features of CodeDeviant include transparently recording program execution data, enabling the programmer to efficiently compare program outputs, and allowing only apt comparisons between executions.

To evaluate the success of our CodeDeviant design, we implemented a prototype CodeDeviant extension for LabVIEW and conducted a laboratory study involving 12 professional LabVIEW programmers. The evaluation compared our CodeDeviant-extended LabVIEW with standard LabVIEW for two key criteria: how accurately the programmers could spot code changes that changed a program's execution behavior and how quickly the programmers could make such decisions.

This work makes the following contributions:

- The findings of formative investigations of programmers refactoring in LabVIEW showing, among other things, the difficulty that programmers had in verifying that their code edits did not alter program behavior.
- A novel tool design based on our formative findings, CodeDeviant, for efficiently detecting behavior-altering changes while integrating seamlessly into programmer workflows.
- A prototype of CodeDeviant implemented as an extension to the LabVIEW visual dataflow programming environment.
- The results of a lab evaluation of CodeDeviant showing that programmers more accurately and more quickly identified behavior-altering code changes with CodeDeviant-extended LabVIEW than with standard LabVIEW.

II. BACKGROUND: VISUAL DATAFLOW LANGUAGES

In this paper, we focus on programmers refactoring visual dataflow code. Unlike textual programming languages (e.g., Java), visual dataflow code consists of boxes (functions) and wires (values). For example, Fig. 1 depicts a small visual dataflow program. Even though visual dataflow languages have a drastically different syntax than textual languages, they have many of the same features, such as modularity.

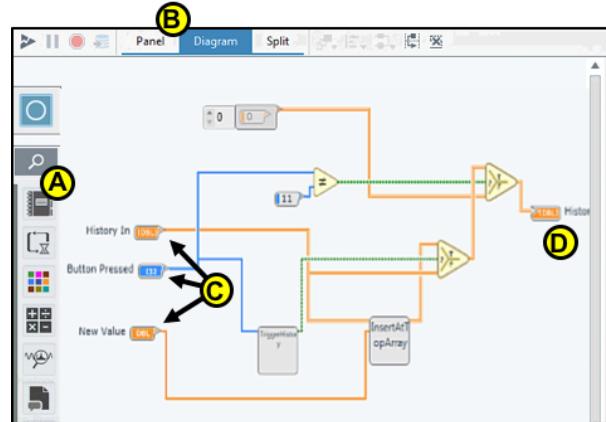


Fig. 1. LabVIEW block-diagram editor. Editors have (A) a palette, along with (B) debugging and other controls. The pictured editor has open a code module with (C) multiple inputs and (D) one output.

In particular, we focus on LabVIEW, a commercially available visual dataflow programming environment that is one of the most widely used visual programming languages [48]. LabVIEW programs are composed of code modules called *Virtual Instruments* (VIs). Fig. 1 illustrates the LabVIEW editor with a particular code module open. This VI has several inputs (e.g., Fig. 1-C) and an output (Fig. 1-D). This example consists of a series of operations performing some logical comparisons (yellow triangles) and calling two other VIs (gray boxes). An important characteristic of VIs is that they can run continuously, allowing multiple rounds of executions to be performed—that is, taking in different inputs and returning different outputs repeatedly before the execution is terminated.

III. FORMATIVE INVESTIGATIONS

To understand the problems that programmers have while refactoring, we performed two formative investigations. In the first, we qualitatively shadowed programmers as they performed refactorings for their jobs, and interviewed them based on our observations. Based on these findings, we began to explore possible solutions, and conducted a second investigation to test the feasibility of one of our candidate solutions.

A. Qualitative Shadowing Study

To better understand how LabVIEW programmers refactor code and the problems they encounter, we performed *qualitative shadowing* [23]. Following this method, a session involved sitting behind a programmer for at least one hour while they worked in their own environment. Our participants were 9 professionals that program every day using LabVIEW. Their job titles consisted of 4 systems engineers, 2 software engineers, 2 application engineers, and 1 hardware engineer. To initiate these shadowing sessions, we emailed the programmers, asking them to arrange a time for us to come to their desk to observe them whenever they planned on “refactoring or cleaning up” code. We observed them while refactoring code, and finished by having a semi-structured interview based on our observations. To elicit feedback on our observations and address follow-up questions, we presented our findings to the same programmers in a group setting.

While working, all nine of the programmers indicated that they have difficulties while refactoring. Based on their verbal remarks while working and their interview responses, a significant issue was that they often introduce bugs while refactoring. One programmer demonstrated an example of this issue: while dragging a few elements around, he accidentally detached a wire from a VI, which changed the program's behavior (but did not produce a compiler error). Another example we observed was when a programmer performed a refactoring, he mixed up two of the outputs, accidentally causing a behavior-altering change (again, without producing a compiler error).

These types of bugs could have potentially been caught by unit tests; however, none of these programmers typically write such tests. In fact, only 3 of the 9 programmers had ever used unit tests in LabVIEW before, and 4 others had only used them in other languages. They indicated that it is too much work to create the unit tests, especially when they do not intend on making changes to the code in the future. One programmer mentioned that if the project uses unit tests, he has to continuously keep the unit tests up to date, which requires too much time for his workflow.

To cope with the difficulties of validating their refactorings, the programmers utilized a variety of strategies. For instance, we observed a programmer using a piece of paper to write down the output of test cases of a VI before he rewrote it. Three other programmers made copies of their entire project prior to making their changes, such that they could run the original version and the modified version simultaneously, to test the behavior. Other times, programmers ran the application to see if the behavior changed, but relied on their recollection of how the program behaved. However, these strategies are error-prone and tedious for the programmer to perform.

B. Feasibility Study

To explore possible solutions for detecting behavior-altering bugs, we analyzed six videos of LabVIEW programmers refactoring code from a prior study [17]. Our goal was to see how often programmers performed refactorings that unintentionally altered the behavior of the program and if that behavior change could be identified by observing the program's output. In the original study, programmers were tasked with refactoring various portions of an existing calculator application.

Each session lasted approximately 90 minutes. First, participants received an introduction to the code base they would be working on. Then, they were instructed that for the next 60 minutes they would be refactoring the code. To help the programmers get started, they were provided with three specific refactorings to perform. Afterwards, they were tasked with continuing to refactor the code however they saw fit. If they got stuck, they were given suggestions of other refactorings. The last 30 minutes of the session involved playing back portions of the video to the participant, and asking questions about each refactoring that they performed.

To analyze whether refactorings altered the program output, we looked at each refactoring episode using the screen-recording video and the participants' talk-aloud data. We

TABLE I
THE REFACTORING EPISODES WE ANALYZED TO SEE IF PROGRAMMERS INTRODUCED BUGS AND IF THE OUTPUT WAS CHANGED. ALTHOUGH P5 DID INTRODUCE ONE BUG, HE DID NOT COMPLETE THE TASK SO IT WAS EXCLUDED FROM OUR ANALYSIS.

Participant	Total refactorings	Buggy refactorings	Behavior-altering
P1	4	2	100%
P2	7	4	100%
P3	5	2	100%
P4	10	2	100%
P5	5	0	--
P6	5	1	100%

considered the refactoring to be buggy if it resulted in behavior that was different than before the change. We inspected the output values to see if the program output could be used to identify the behavior-altering change. For this analysis, we assumed that the programmer would have executed the program before and after the change using the same input.

As shown in Table I, participants often introduced bugs while refactoring. In fact, all but one participant performed buggy refactorings, and on average, 31% of their refactorings were buggy. Furthermore, every bug they introduced caused the output of the program to change. A particularly common bug was wiring code elements incorrectly (e.g., swapping two wires). Other bugs included not handling corner cases for rewritten code (e.g., if the input is zero), changing a value in some locations but not all the locations, incorrectly initializing variables that were moved out of a loop or inner VI, and extracting a method but not calling it correctly. Understanding these behavior-altering changes were an initial step towards designing a tool that could detect these changes.

IV. TOOL DESIGN

To address the problems programmers have in detecting behavior changes after refactoring, we designed the CodeDeviant tool for visual dataflow programming environments. CodeDeviant enables programmers to compare the program output of a previous execution to the current execution so they can determine if their refactoring had unintended side effects on program behavior. By providing this information to programmers, CodeDeviant aims to enable programmers to test their changes more quickly and more accurately.

Based on our qualitative shadowing observations and feedback provided by programmers, we conceived of three key design principles for CodeDeviant:

- Transparently record program execution data as executions are performed by the programmer. In particular, CodeDeviant records the input values, output values, and metadata (e.g., timestamp) for each execution.
- Enable the programmer to efficiently compare selected program executions from the recorded history to see if the

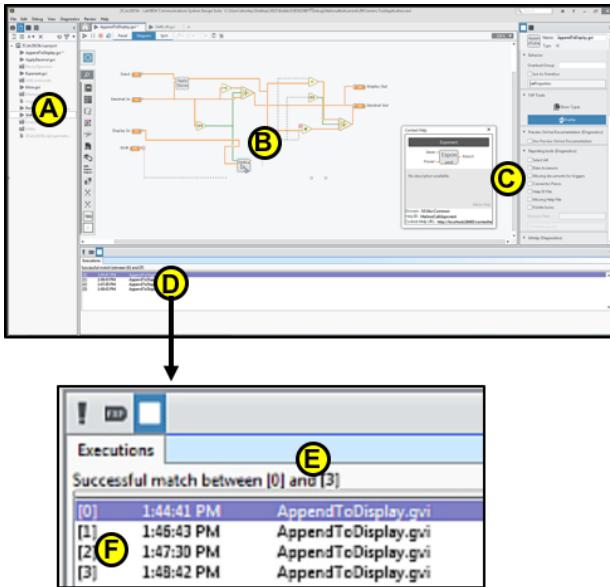


Fig. 2. CodeDeviant-extended LabVIEW IDE. The standard LabVIEW IDE features include (A) a project explorer, (B, see also Fig. 1) the code editor, and (C) contextual help and properties. CodeDeviant extends the IDE with (D) an additional pane that provides (F) a history of executions and that indicates (E) whether the behavior has changed.

program outputs changed while integrating seamlessly into their workflow.

- Only allow apt comparisons between executions and notify the user if there is not a suitable comparison. For example, if an input parameter is removed, it is not clear how CodeDeviant would compare the values.

To evaluate our design, we implemented CodeDeviant as an extension to the LabVIEW development environment. The remainder of this section explains the specific features of our CodeDeviant extension that satisfy these principles.

A. Transparently Record Program Executions

In our LabVIEW implementation, whenever a programmer executes a code module (i.e., a VI), CodeDeviant automatically records the sets of input and corresponding output values as well as metadata (e.g., timestamp and information about the VI). Recording is done transparently, not requiring any explicit user action. To implement this feature, we leveraged LabVIEW’s compiler framework to walk the abstract syntax tree of the VI to identify the input and output nodes (recall Fig. 1-C,D). We then utilized LabVIEW’s existing runtime framework, which already has highly optimized features for asynchronously logging high volumes of streaming data to a file. Being able to handle streaming data is an important criteria since LabVIEW is often used to stream vast amounts of data from instruments (e.g., an oscilloscope).

B. Efficiently Compare Program Executions

The main goal of CodeDeviant is to efficiently compare program executions to detect whether the behavior has changed. After an execution is finished, an entry is added to the history pane, depicted in Fig. 2-F, which displays the VI name and timestamp of when it was executed. To perform a comparison,

the user selects which prior execution to use as a point of comparison by clicking the associated row in the history listing. Then, the next time the application is executed, the current execution will be compared to the selected execution.

To compare the executions, CodeDeviant inspects the logged input and output values. It first performs an intersection of the input values from both executions (and ignores any values that were not present in both executions). This step is needed because unlike functions in most textual languages, a VI can execute continuously, either streaming data from hardware or acting as a long-running interactive system. For this reason, VIs can be continuously fed inputs and continuously provide outputs—unlike in Java, for example, where a function call takes in one set of arguments and returns one value. CodeDeviant then compares whether the outputs are equivalent between executions, given the same inputs. For example, if execution A has two input/output pairs ((2,8), (5,20)) and execution B has three pairs ((3,12), (5,11), (8,8)), CodeDeviant would do an intersection on the inputs values of A and B, which in this case contains only the input value 5. The next step is to compare the corresponding outputs given 5 as input, which in this case, do not match ($20 \neq 11$).

Once CodeDeviant compares the executions, it will notify the programmer whether the behavior matches between the selected execution and the most recent execution (Fig. 2-E). Once the programmer gets feedback from the tool, he or she can then manually inspect the code if necessary to find the cause of the behavior change.

To better fit into programmers’ workflows, CodeDeviant allows for comparisons without any explicit interactions to do so. If the programmer does not choose an execution in the history pane, it will default to the oldest execution in the current development session. This execution was chosen as the default to ensure that it came before the code change. Additionally, CodeDeviant allows for repeated comparisons without any additional actions. That is, the programmer can execute the application over and over, using either the same selected execution to compare with, or by selecting another execution in the history.

C. Allow Only Apt Comparisons

As the programmer is performing changes and executing the application, CodeDeviant allows for only apt comparisons since it is possible that there may not be a reasonable way to compare the current execution with the selected previous execution. For example, if the programmer modifies the inputs (e.g., adds an additional parameter), it is not clear how CodeDeviant should compare the executions, and CodeDeviant will provide an error message. Similarly, if the executions do not share any input values, CodeDeviant cannot determine if the behavior is the same and will notify the user.

V. EVALUATION METHOD

To investigate how effectively CodeDeviant helps programmers in detecting behavior-altering refactorings, we ran a within-subjects lab study of programmers refactoring and

validating their refactorings. Each participant received two treatments, the control treatment, the standard LabVIEW environment, and the CodeDeviant treatment, an extended version of LabVIEW with CodeDeviant features enabled.

The research questions that we addressed with our empirical evaluation of CodeDeviant were as follows:

- RQ1: Do programmers using CodeDeviant-extended LabVIEW find behavior-altering bugs more *accurately* than programmers using standard LabVIEW?
- RQ2: Do programmers using CodeDeviant-extended LabVIEW find behavior-altering bugs more *quickly* than programmers using standard LabVIEW?
- RQ3: Do programmers consider CodeDeviant to be *helpful*?

Our participants consisted of 12 professional LabVIEW programmers (11 male, 1 female) from National Instruments. They reported, on average, 4.58 years of programming experience ($SD = 1.73$) and 2.23 years of LabVIEW experience ($SD = 1.93$). All participants reported programming in LabVIEW as part of their daily work.

As their primary tasks, the participants performed two refactorings on an existing calculator application written in LabVIEW. They were also asked to verify whether or not the refactoring changed the behavior of the application (but not to perform additional fixes). Each task was based on a refactoring from Fowler's catalog of refactorings [12]. The first task involved replacing two blocks of code with a built-in function (similar to Fowler's Replace Algorithm refactoring). The built-in function behaved differently than the blocks, and thus, in validating the change, the correct answer was that it does change the behavior. The second task involved performing an Extract Method refactoring which should not have changed the program's behavior.

Each session lasted no more than 30 minutes. First, all participants filled out a background questionnaire, and received an introduction to the latest version of LabVIEW and the calculator application. Next, the participants performed the two refactoring tasks where they were free to modify and test the code however they saw fit. Each participant received one treatment for the first task and the other for the second task. Half of the participants were randomly selected to use CodeDeviant-extended LabVIEW first, and the other half used standard LabVIEW first. We asked each participant to "think aloud" as he/she worked. At the end of the session, participants answered a questionnaire regarding the tool and took part in a semi-structured interview. As data, we recorded audio and screen-capture video of each session.

VI. EVALUATION RESULTS

A. RQ1 Results: Accuracy of Detecting Bugs

As Fig. 3 shows, when participants used CodeDeviant, they correctly assessed whether their refactorings resulted in changes to the program's behavior far more often than when they used only standard LabVIEW. In fact, when using CodeDeviant, everyone provided the correct answer for the task. In contrast, when using standard LabVIEW, only a third

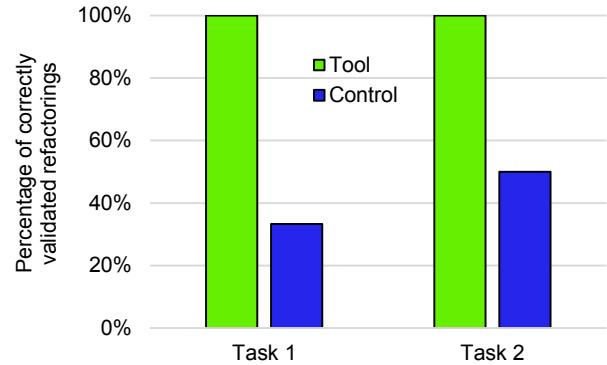


Fig. 3. CodeDeviant users were significantly more accurate in validating their refactorings.

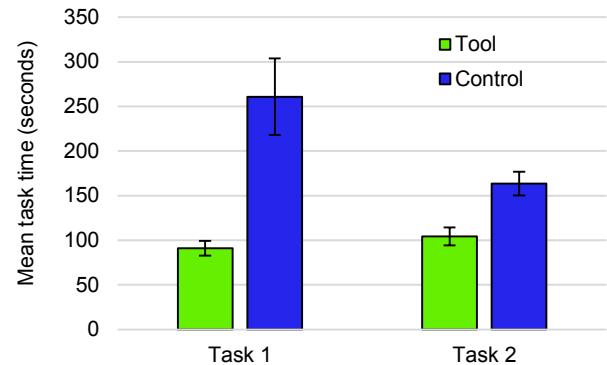


Fig. 4. CodeDeviant users were significantly faster in performing and validating the refactorings (smaller bars are better). Whiskers denote standard error.

of the participants provided the correct answer for the first task and only half for the second task. The differences were significant for Task 1 ($\chi^2(1, N = 12) = 6, p = 0.01$) and Task 2 ($\chi^2(1, N = 12) = 4, p < 0.05$).

B. RQ2 Results: Time on Task

As Fig. 4 shows, When participants used CodeDeviant, they also completed the tasks considerably faster than when they used only standard LabVIEW. For Task 1, participants using CodeDeviant took roughly a third of the time taken by those using standard LabVIEW. For Task 2, participants using CodeDeviant completed Task 2 over 40% faster than those using standard LabVIEW. A Mann-Whitney U test showed significance for both Task 1 ($U = 0, Z = 2.9, p = 0.003$) and Task 2 ($U = 2, Z = 2.5, p = 0.01$).

C. RQ3 Results: Opinions of the Participants

As Fig. 5 shows, the participants generally considered CodeDeviant to be helpful and would use it for their everyday work. Only one participant responded that the tool was not helpful. Furthermore, only two participants said they would not use CodeDeviant if they had it available to them. Details of their concerns are described in the Discussion section.

VII. DISCUSSION

Overall, the results of our CodeDeviant evaluation were notably positive. Participants using CodeDeviant identified

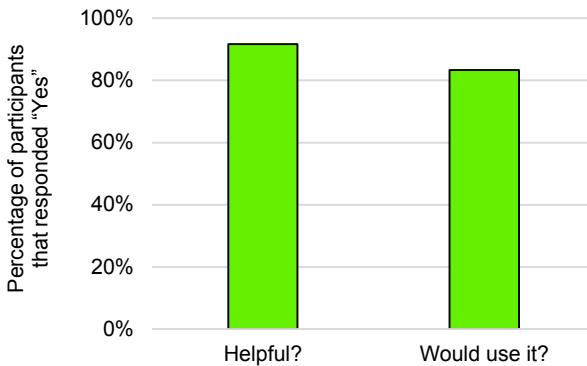


Fig. 5. Participant responses were highly positive on the CodeDeviant opinion questionnaire (“Yes” or “No” answers).

behavior-altering bugs significantly more accurately than those using the control treatment. Moreover, CodeDeviant also helped participants complete the refactoring tasks significantly faster than did the control treatment. In addition to the positive results for these objective performance measures, the participants also expressed predominantly favorable subjective opinions of CodeDeviant.

A. Qualitative Observations

To better understand the reasons behind our overwhelmingly positive results, we analyzed our data for qualitative evidence to help explain these outcomes. In particular, we reviewed the participants’ comments and mapped from quantitative data points to qualitative episodes of participant behavior, examining those episodes in detail to help explain and expand upon our results.

1) Why Participants Validated Changes More Accurately with CodeDeviant (RQ1): Every participant using CodeDeviant had 100% accuracy in detecting behavior-altering bugs. For every task, participants receiving the CodeDeviant treatment used CodeDeviant to validate their changes, and CodeDeviant reported correctly whether or not the program’s output had changed. For example, during P3’s second task, he identified the relevant code that needed to be refactored, so he then executed the AppendToDisplay VI twice with two different sets of input. He then performed the code change and reran the VI. Finally, he turned to CodeDeviant, which correctly reported that the behavior did not change.

However, the participants had a much more difficult time when they did not use CodeDeviant. Multiple participants ran the VI one or more times before and after their changes to see if the program’s output changed. For example, P5 deliberately executed the program with one set of inputs before making his edits:

P5: “If I care about testing this, then I should go do that first.”

Once he finished his change, he reran the application. Although the output was visibly different, he mistakenly declared that it worked the same, perhaps unable to fully recall the original output. Participants P4, P6, and P9 followed similar strategies, and were also unsuccessful in noticing changes in the output of their programs. In one extreme case, P4 alternated between viewing the code and running the application

five times prior to making his change, and still failed to notice that his program’s output had changed.

Even when participants tried other strategies to validate their changes, they still had difficulties. Participant P10 took a more thorough approach to testing the program by writing down the program output on a piece of paper. However, this note-taking strategy was ultimately unsuccessful. She incorrectly thought that the program’s behavior had changed, perhaps because she failed to notice that she had changed the input values between her runs (an inconsistency that CodeDeviant would have caught). In contrast, participant P12 did not rely on running the application at all. Instead, he examined the code, following nearly every wire in RemoveDecimalFromDisplay and ApplyDecimal to understand his change. After examining the wires for over two minutes, he finally declared:

P12: “I’m confident this code does the same thing.”

Unfortunately, he was incorrect: the behavior had changed.

2) Why Participants Validated Changes Faster with CodeDeviant (RQ2): Not only were participants more accurate while using CodeDeviant, they also completed the tasks considerably faster. This speedup is likely thanks to the automation provided by CodeDeviant, which eliminated the need to use tedious manual change-validation strategies, such as testing and trying to remember the output values produced before the change, or tracing wires in the hopes of discovering a bug. Using CodeDeviant, participant P10 was able to achieve the fastest overall time for the first task. She began by testing the application, performed the change, ran the application once, and consulted the CodeDeviant output. The whole process took only 85 seconds. Similarly, participant P7 completed the second task in just a little over a minute using CodeDeviant. Thanks to CodeDeviant, he spent the majority of this time on making edits to the code, rather than, say, testing it.

In contrast, when using standard LabVIEW without CodeDeviant, participants took much longer in completing their tasks. The participants’ strategies for validating their code seemed to be the main cause of this slowdown. For example, in the case of participant P8’s first task, he simply stared at the code for long stretches of time without saying or doing anything. Although his strategy was not entirely clear, he may have been visually tracing the code relevant to his change. He was ultimately correct in reporting that the code change had altered the program’s behavior, but it took him over 7 minutes to reach that conclusion.

3) Why Participants Liked (or Disliked) CodeDeviant (RQ3): Participants were generally favorable of CodeDeviant. In the interview, participants expanded on their thoughts of CodeDeviant, providing a range of feedback. Several participants explained how CodeDeviant alleviated the burden of remembering the program behavior as they worked. In particular, participant P11 described his normal working environment, where he can write down his test inputs and outputs on a piece of paper, and how using CodeDeviant will make that process more efficient:

P11: “On the one where I was doing it without having the tool, it isn’t terribly difficult to write down... when I’m sitting at

my desk I have notepads and pens, but if you can get around having to have one... If you have one output it's fine, but if you have something that has to output an entire array, being able to validate that [with the tool] is really nice."

Other participants had similar sentiments regarding how CodeDeviant can enhance efficiency:

P6: "Now [without the tool] you have to manually see if the output is the same. If you have more inputs or outputs it is harder to do it manually. If you are working on something complex, this would be a really useful tool."

P8: "[Without the tool] it was all on me to remember what I got for the outputs. It isn't too terrible for a small VI but as soon as you hit any level of complexity..."

Furthermore, participants elaborated on the general usefulness of the tool in regards to testing. When asked why they found CodeDeviant useful, P3 and P7 expressed the importance of testing, which they believed CodeDeviant helped with:

P3: "If you don't have to do all the setup yourself, and you just have a tool that will do it for you then I feel like more people would be more willing to do it."

P7: "Having any kind of testing is incredibly important."

Although most participants were favorable of CodeDeviant, one participant said it was not helpful, and two said they would not use it in their daily workflows. Participant P1 reported his concerns about choosing the correct input values:

P1: "What if it only works because these inputs are the ones I'm testing but my change doesn't work."

His concern is valid, but this problem also exists without the tool (e.g., manually providing inputs and observing the outputs). Participant P9 reported that he believed the tool was helpful but would not use it because he believed it might be difficult for others to discover the feature and to integrate into their workflow.

P9: "I think it is a thing that could be useful to people who know about it and are trained to do that, and see the benefit of it... It is kind of a tricky situation to figure out how to help people who don't know how to use the tools that exist."

He later explained that it could be integrated more closely into a programmer's typical workflow:

P9: "It would be neat if it just showed up as a warning in their errors window."

B. Opportunities for Improving CodeDeviant

Based on our participants' feedback and the findings from the user study, we identified several key opportunities for potentially improving the design of CodeDeviant.

1) Interaction Design Improvements: One key opportunity for improvement is to better explain to the programmer how executions behaved differently. Currently, CodeDeviant reports whether there is a difference in behavior, but does not communicate how it differed or by how much. For example, CodeDeviant could display the specific input values that resulted in different output values between executions. To provide the programmer more context about what was tested, CodeDeviant could report a correctness percentage (e.g., 75% of the tested values are equal) as well as a coverage percentage (e.g., 20% of the original inputs were tested).

2) Performance Overhead Reduction: A second key opportunity for improvement is by reducing the performance overhead of CodeDeviant. The main overhead stems from recording the program output at runtime. (We did not detect any noticeable performance impact in CPU load while running CodeDeviant.) In our test cases, VIs that ran only once used little storage for recordings (<1KB). However, VIs that ran continuously (certain GUIs and data acquisition functions) used as much as 50MB of storage per minute of recording. While inspecting these data, we observed that over 90% were redundant, and could be filtered periodically to save space.

3) Extended Coverage of Program Behaviors: A third opportunity for improvement is to expand the program behaviors that can be compared by CodeDeviant. Although CodeDeviant never failed to detect a behavior change for all of our participants, there are possible scenarios where it could fail. In particular, any application where it is difficult to reproduce the same input values could be problematic. For example, if the application acquires streaming data from specialized hardware (e.g., an instrument for real-time radio measurements) such that each execution will not yield the same input values (or some subset thereof), then CodeDeviant will not be able to do a comparison. To address this problem, CodeDeviant could be enhanced with *replay debugging*, a technique that enables the replaying of events that produced a particular outcome [29].

VIII. THREATS TO VALIDITY

Our evaluation study has several threats to validity that are inherent to laboratory studies of programmers. The code base was small, and thus, may not be representative of all programs; however, to enhance its realism, we based it on an open source LabVIEW project. Our participants may not have been representative of all expert programmers; however, they were all professional LabVIEW programmers. Reactivity effects, such as the participant trying to please the researchers, may have occurred; however, we attempted to minimize these effects by presenting the two versions of LabVIEW to participants as possible design alternatives, and by not revealing that the CodeDeviant version was the researchers' creation. Finally, our sample size was small, with only 12 participants performing 2 tasks each; however, we used multiple metrics and both quantitative and qualitative analyses to triangulate and enhance confidence in our findings.

IX. RELATED WORK

A. Refactoring Support

Researchers have proposed a variety of automated refactoring tools to improve the efficiency and correctness of these code changes, but they rely on programmers explicitly utilizing these features. However, many studies have shown that the majority of refactorings are performed manually [13], [19], [28], [30], [44]. One notable tool, SafeRefactor [41], generates unit tests for the original code and the refactored code to verify that the behavior does not change when applying automated refactorings. Other work in automated refactoring has been to formally verify the refactoring operations (e.g., [9], [25]) and

the refactoring engine (e.g., [26], [40], [42]), and yet there are still bugs introduced by the most commonly used refactoring tools [4], [39], [41], [44], [47]. CodeDeviant was designed to fit into programmers' existing workflow, without requiring them to use automated refactorings.

To better integrate refactoring tools into programmers' existing workflows, researchers have designed tools to assist in manual refactorings, but they rely on detecting when a manual refactoring has occurred. For example, BeneFactor [13], GhostFactor [14], and WitchDoctor [11] aim to recognize when a programmer is performing a refactoring manually and provide features to automatically complete the change while attempting to validate correctness. RefDistiller [1] takes a different approach by identifying manual refactorings after they are completed, and provides features for the programmer to review these changes, while suggesting missing changes and extra changes that are needed to maintain the same behavior. However, these tools rely on static analysis to identify manual refactorings and are limited in the types of refactorings that they support, unlike CodeDeviant, which does not need to detect that the programmer is refactoring.

B. Change Impact Analysis

Change impact analysis is a complementary approach to CodeDeviant's, which locates portions of the code that may be affected by a code change [22]. These analyses could be leveraged by tools to identify whether a refactoring is behavior altering. However, a variety of issues have prevented such techniques from being adopted by programmers in practice. For example, these tools output a list of code locations that are potentially impacted by a change (e.g., EAT [2], Sieve [38], Jimpa [6], JDIA [18], Impala [16], JRipples [5], and ROSE [49]), which then requires a programmer to manually investigate these locations. Another issue is that they can have high rates of false positives and false negatives [16], while more accurate algorithms incur substantial performance costs (e.g., PathImpact [32]). Yet another barrier of these tools is that they may require substantial effort to setup and maintain, such as creating unit test suites (e.g., Crisp [8]) or instrumenting the source code prior to use (e.g., EAT [2]).

C. Program Steering

A related idea to comparing program outputs to validate a code change, is *program steering* [3], which provides continuous feedback to the programmer and allows the programmer to modify the program at runtime. For example, Forms/3 provides affordances to *time travel*, so the programmer can investigate the causes and effects of a program's behavior [3], which could potentially help a programmer detect a bug in their refactoring. Since these features were implemented in a spreadsheet-like environment, it is an open question how well they would generalize to a visual dataflow programming environment (e.g., with streaming data). Additionally, program steering features require considerable changes to a programmer's workflow, which may be a barrier to adoption.

X. CONCLUSION

In this paper, we have presented the novel CodeDeviant tool design to support programmers in detecting behavior-altering bugs while refactoring visual dataflow code. An evaluation study comparing the CodeDeviant-extended LabVIEW with the standard LabVIEW IDE made the following key findings:

- RQ1 (accuracy): Programmers using CodeDeviant-extended LabVIEW identified behavior-altering bugs significantly more accurately than with standard LabVIEW.
- RQ2 (time): Programmers using CodeDeviant-extended LabVIEW completed the refactoring tasks significantly faster than with standard LabVIEW.
- RQ3 (user opinions): Programmers generally found CodeDeviant helpful and agreed that they would use it in their daily work.

We hope that CodeDeviant and our findings represent a noteworthy advancement toward helping programmers refactor their code more correctly and efficiently. Moving beyond our current work, a promising direction for the future is to explore novel ways in which a programmer can effectively compare all aspects of their program to some previous state of the program, including input values, output values, and code changes. Although there have been tools proposed that provide specific comparisons (e.g., *how did my code look previously?* [17]), there has not been a comprehensive system that allows the programmer to compare all aspects of their program and its behavior. Our CodeDeviant design and implementation demonstrated the strong potential of such a system, eliciting extensive positive feedback and optimism from professional programmers. We believe this work represents a substantial step toward better supporting programmers in the fundamental, yet tedious and error-prone, task of refactoring code.

ACKNOWLEDGMENTS

We give special thanks to Andrew Dove for his counsel on all things LabVIEW. This material is based upon work supported by National Instruments and by the National Science Foundation under Grant No. 1302117. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of National Instruments or of the National Science Foundation.

REFERENCES

- [1] E. L. G. Alves, M. Song, and M. Kim, "RefDistiller: A refactoring aware code review tool for inspecting manual refactoring edits," in *Proc. 22nd ACM SIGSOFT Int'l Symp. Foundations of Software Engineering (FSE '14)*, 2014, pp. 751–754.
- [2] T. Apivattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," in *Proc. 27th Int'l Conf. Software Engineering (ICSE '05)*, 2005, pp. 432–441.
- [3] J. W. Atwood, M. M. Burnett, R. A. Walpole, E. M. Wilcox, and S. Yang, "Steering programs via time travel," in *Proc. 1996 IEEE Symp. Visual Languages*, Sep 1996, pp. 4–11.
- [4] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs?: An empirical study," in *Proc. 2012 IEEE 12th Int'l Working Conf. Source Code Analysis and Manipulation (SCAM '12)*, 2012, pp. 104–113.

- [5] J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich, "JRipples: A tool for program comprehension during incremental change," in *13th Int'l Workshop on Program Comprehension (IWPC'05)*, May 2005, pp. 149–152.
- [6] G. Canfora and L. Cerulo, "Impact analysis by mining software and change request repositories," in *11th IEEE Int'l Software Metrics Symposium (METRICS '05)*, Sept 2005, pp. 21–29.
- [7] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko, "Let's go to the whiteboard: How and why software developers use drawings," in *Proc. SIGCHI Conf. Human Factors in Computing Systems (CHI '07)*, 2007, pp. 557–566.
- [8] O. C. Chesley, X. Ren, and B. G. Ryder, "Crisp: A debugging tool for Java programs," in *21st IEEE Int'l Conf. Software Maintenance (ICSM'05)*, Sept 2005, pp. 401–410.
- [9] M. Cornelio, A. Cavalcanti, and A. Sampaio, "Sound refactorings," *Science of Computer Programming*, vol. 75, no. 3, pp. 106–133, 2010.
- [10] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Proc. the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symp. The Foundations of Software Engineering (ESEC-FSE '07)*, 2007, pp. 185–194.
- [11] S. R. Foster, W. G. Griswold, and S. Lerner, "WitchDoctor: IDE support for real-time auto-completion of refactorings," in *Proc. 2012 Int'l Conf. Software Engineering*, 2012, pp. 222–232.
- [12] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.
- [13] X. Ge, Q. L. DuBose, and E. Murphy-Hill, "Reconciling manual and automatic refactoring," in *Proc. 34th Int'l Conf. Software Engineering (ICSE '12)*, 2012, pp. 211–221.
- [14] X. Ge and E. Murphy-Hill, "Manual refactoring changes with automated refactoring validation," in *Proc. 36th Int'l Conf. Software Engineering (ICSE '14)*, 2014, pp. 1095–1105.
- [15] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131–174, 1996.
- [16] L. Hattori, G. dos Santos Jr, F. Cardoso, and M. Sampaio, "Mining software repositories for software change impact analysis: A case study," in *Proc. 23rd Brazilian Symp. Databases SBBD '08*, 2008, pp. 210–223.
- [17] A. Z. Henley and S. D. Fleming, "Yestercode: Improving code-change support in visual dataflow programming environments," in *2016 IEEE Symp. Visual Languages and Human-Centric Computing (VL/HCC '16)*, Sept 2016, pp. 106–114.
- [18] L. Huang and Y. T. Song, "A dynamic impact analysis approach for object-oriented programs," in *2008 Advanced Software Engineering and Its Applications*, Dec 2008, pp. 217–220.
- [19] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proc. ACM SIGSOFT 20th Int'l Symp. the Foundations of Software Engineering (FSE '12)*, 2012, pp. 50:1–50:11.
- [20] A. Kittur, A. M. Peters, A. Diriyeh, T. Telang, and M. R. Bove, "Costs and benefits of structured information foraging," in *Proc. SIGCHI Conf. Human Factors in Computing Systems*, ser. CHI '13, 2013, pp. 2989–2998.
- [21] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi, "A case study in refactoring a legacy component for reuse in a product line," in *Software Maintenance, 2005. ICSM'05. Proc. 21st IEEE Int'l Conf.*, Sept 2005, pp. 369–378.
- [22] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," *Software Testing, Verification and Reliability*, vol. 23, no. 8, pp. 613–646, 2013.
- [23] S. McDonald, "Studying actions in context: a qualitative shadowing method for organizational research," *Qualitative Research*, vol. 5, no. 4, pp. 455–473, 2005.
- [24] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, Feb 2004.
- [25] T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens, "Formalizing refactorings with graph transformations," *J. Softw. Maint. Evol.*, vol. 17, no. 4, pp. 247–276, 2005.
- [26] M. Mongiovì, "Safira: A tool for evaluating behavior preservation," in *Proc. ACM Int'l Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*, 2011, pp. 213–214.
- [27] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi, "Does refactoring improve reusability?" in *Int'l Conf. Software Reuse*. Springer, 2006, pp. 287–297.
- [28] E. Murphy-Hill, C. Parnin, and A. Black, "How we refactor, and how we know it," *Software Engineering, IEEE Transactions on*, vol. 38, no. 1, pp. 5–18, Jan 2012.
- [29] S. Narayanasamy, G. Pokam, and B. Calder, "BugNet: Continuously recording program execution for deterministic replay debugging," in *Proc. 32Nd Annual Int'l Symp. Computer Architecture (ISCA '05)*, 2005, pp. 284–295.
- [30] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, "A comparative study of manual and automated refactorings," in *Proc. 27th European Conf. Object-Oriented Programming (ECOOP '13)*, 2013, pp. 552–576.
- [31] W. F. Opdyke, "Refactoring: An aid in designing application frameworks and evolving object-oriented systems," in *Proc. of 1990 Symp. Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, 1990.
- [32] A. Orso, T. Apivattanapong, J. Law, G. Rothermel, and M. J. Harrold, "An empirical comparison of dynamic impact analysis algorithms," in *Proc. 26th Int'l Conf. Software Engineering (ICSE '04)*, 2004, pp. 491–500.
- [33] N. Pennington, "Stimulus structures and mental representations in expert comprehension of computer programs," *Cognitive Psychol.*, vol. 19, no. 3, pp. 295–341, 1987.
- [34] D. Piorkowski, S. Fleming, C. Scaffidi, C. Bogart, M. Burnett, B. John, R. Bellamy, and C. Swart, "Reactive information foraging: An empirical investigation of theory-based recommender systems for programmers," in *Proc. ACM SIGCHI Conf. Human Factors in Computing Systems*, ser. CHI '12, 2012, pp. 1471–1480.
- [35] D. J. Piorkowski, S. D. Fleming, I. Kwan, M. M. Burnett, C. Scaffidi, R. K. Bellamy, and J. Jordahl, "The whats and hows of programmers' foraging diets," in *Proc. SIGCHI Conf. Human Factors in Computing Systems*, ser. CHI '13, 2013, pp. 3063–3072.
- [36] J. U. Pipka, "Refactoring in a test first world," in *Proc. Third Int'l Conf. eXtreme Programming and Flexible Processes in Software Eng.*, 2002.
- [37] N. Rachatasumrit and M. Kim, "An empirical investigation into the impact of refactoring on regression testing," in *2012 28th IEEE Int'l Conf. Software Maintenance (ICSM)*, Sept 2012, pp. 357–366.
- [38] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Sieve: A tool for automatically detecting variations across program versions," in *21st IEEE/ACM Int'l Conf. Automated Software Engineering (ASE '06)*, Sept 2006, pp. 241–252.
- [39] M. Schäfer and O. de Moor, "Specifying and implementing refactorings," in *Proc. ACM Int'l Conf. Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*, 2010, pp. 286–301.
- [40] M. Schäfer, T. Ekman, and O. de Moor, "Sound and extensible renaming for Java," in *Proc. 23rd ACM SIGPLAN Conf. Object-oriented Programming Systems Languages and Applications (OOPSLA '08)*, 2008, pp. 277–294.
- [41] G. Soares, R. Gheyi, D. Serey, and T. Massoni, "Making program refactoring safer," *IEEE Software*, vol. 27, no. 4, pp. 52–57, July 2010.
- [42] G. Soares, R. Gheyi, and T. Massoni, "Automated behavioral testing of refactoring engines," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 147–162, Feb. 2013.
- [43] K. T. Stolee and S. Elbaum, "Refactoring pipe-like mashups for end-user programmers," in *Proc. 33rd Int'l Conf. Software Engineering (ICSE '11)*, 2011, pp. 81–90.
- [44] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, "Use, disuse, and misuse of automated refactorings," in *Proc. 34th Int'l Conf. Software Engineering (ICSE '12)*, 2012, pp. 233–243.
- [45] F. Vonken and A. Zaidman, "Refactoring with unit testing: A match made in heaven?" in *Proc. 2012 19th Working Conf. Reverse Engineering (WCWRE '12)*, 2012, pp. 29–38.
- [46] P. Weissgerber and S. Diehl, "Identifying refactorings from source-code changes," in *21st IEEE/ACM Int'l Conf. Automated Software Engineering (ASE '06)*, Sept 2006, pp. 231–240.
- [47] P. Weissgerber and S. Diehl, "Are refactorings less error-prone than other changes?" in *Proc. 2006 Int'l Workshop on Mining Software Repositories (MSR '06)*, 2006, pp. 112–118.
- [48] K. N. Whitley, L. R. Novick, and D. Fisher, "Evidence in favor of visual representation for the dataflow paradigm: An experiment testing LabVIEW's comprehensibility," *Int'l Journal of Human-Computer Studies*, vol. 64, no. 4, pp. 281–303, 2006.
- [49] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 429–445, June 2005.

End-User Development in Social Psychology Research: Factors for Adoption

Daniel Rough

SACHI Research Group

University of St Andrews

Email: djr53@st-andrews.ac.uk

Aaron Quigley

SACHI Research Group

University of St Andrews

Email: aquigley@st-andrews.ac.uk

Abstract—Psychology researchers employ the Experience Sampling Method (ESM) to capture thoughts and behaviours of participants within their everyday lives. Smartphone-based ESM apps are increasingly used in such research. However, the diversity of researchers' app requirements, coupled with cost and complexity of their implementation, has prompted end-user development (EUD) approaches. In addition, limited evaluation of such environments beyond lab-based usability studies precludes discovery of factors pertaining to real-world EUD adoption. We first describe the extension of *Jeeves*, our visual programming environment for ESM app creation, in which we implemented additional functional requirements, derived from a survey and analysis of previous work. We further describe interviews with psychology researchers to understand their practical considerations for employing this extended environment in their work practices. Results of our analysis are presented as factors pertaining to the adoption of EUD activities within and between communities of practice.

I. INTRODUCTION

The Experience Sampling Method (ESM) is a methodology employed primarily in psychology and clinical research to survey participants in-the-moment, in their natural contexts [1], [2]. ESM has a number of advantages over retrospective surveys; primarily, it has high ecological validity as participants are assessed in their natural environments, as opposed to a lab or clinic. Further, it minimises recall bias, as participants' experiences are captured in situ. As a recent example of such advantages, Lenaert et al. demonstrate how ESM can be used to identify fluctuating emotions in patients with Acquired Brain Injury, to improve researchers' understanding and consequent treatment [3].

Although smartphones are an ideal medium for delivery of ESM surveys, previous work has recognised that researchers face significant programming barriers to adoption of smartphone-based ESM, given that the necessary software development skills are often outside their area of expertise [4], [5]. Short of defaulting to paper-based methods, researchers must rely on professional support to develop custom apps for their studies. As well as the expense of this approach, bespoke apps are inflexible to diverse, dynamic requirements of researchers and their participants. With respect to these issues, end-user development (EUD) presents a possible solution [6].

EUD tools for ESM apps have previously been developed in response to this need, which we survey in Section II. However, despite calls to investigate the socio-technical aspects of EUD introduction, there is little research on the impact of such technology in working practices. Thus we ask the following research question: *What factors influence the adoption of technology for psychology researchers to develop experience sampling smartphone apps?*

In this paper we make two research contributions, centred on the design of an extensible EUD tool for ESM app creation (hereby referred to as an EUD-ESM tool). We first survey the potential utility of ESM apps and derive requirements towards facilitating social psychology research, and we describe the supporting design decisions implemented in the novel extension of *Jeeves*, our existing EUD-ESM tool [7].

As a second contribution, we address the paucity of research into adoption requirements for EUD in working practices, through qualitative analysis of psychology researchers' perceptions of *Jeeves*, and EUD practice in general, obtained through semi-structured interviews. Analysing our results with respect to current models of technology acceptance, we discuss the necessary factors for adoption of an EUD-ESM tool in social psychology, and how these relate to general “public-outward EUD”, where one community of end-users develops for another [8], thus broadening the implications of this study.

II. RELATED WORK

Research relevant to the goals of our work covers three areas - potential utility of ESM, existing EUD-ESM tools, and adoption of EUD in professional practice. In a traditional model of ESM development, researchers express their requirements to a programmer, who creates a bespoke app for that particular study. Participants install the app and return to researchers after the study period. In our proposed update to this model (Figure 1), these stakeholders have separate development roles, represented by the three levels of tailoring defined by Mørch [9]. The programmer's role is that of a *meta-designer*, who creates the components necessary for the researcher through **extension**. Using these components, the researcher can **create** or **modify** apps, with behaviour that can be **customised** by study participants. Beyond development paradigms that are usable by non-programmers, EUD encompasses socio-technical aspects of software. Fischer's

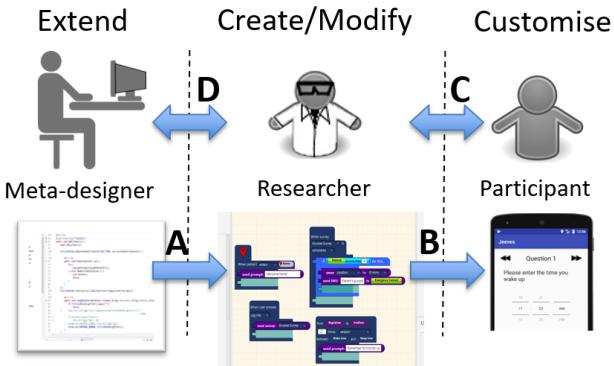


Fig. 1. Our meta-design model of ESM app development, where stakeholders collaborate and perform EUD activities “during use”

meta-design framework acknowledges that development is a continuous effort that involves ongoing collaboration [10].

A. Potential Utility of ESM

A review of recent literature highlights two application areas of repeated, longitudinal assessment inherent in ESM:

- 1) In **research**, to investigate how participant variables fluctuate over time and in different contexts
- 2) In **practice**, to allow participants to independently monitor aspects of their mental health

Previous work has surveyed the benefits of ESM to behavioural research [4], [5], and the process of in-situ, longitudinal assessment and intervention has been thoroughly discussed as an asset to participants’ health in practical applications [11], [12]. Additionally, general mobile health (mHealth) interventions have been surveyed [13], providing evidence-based recommendations for features relevant to ESM, such as self-monitoring. With these applications in mind, our related work summarises potential beneficial features of smartphone ESM to both researchers, and participants themselves.

Objective Context Triggering: Automated capture and inference of objective sensor data can support the delivery of assessments, interventions, or reminders at ideal times, introducing new possibilities for psychology research. For example, contextual sensing can be used to predict the “interruptibility” of participants, to ensure surveys are sent at minimally intrusive moments [14]. In a medical context, participants’ mood was inferred from sensor values to deliver tailored feedback in an intervention for depression [15]. Location-based reminders have been endorsed as a useful feature in behavioural interventions [16], and feedback on smartphone-sensed activity can also promote self-awareness [17].

Subjective Context Triggering: A deliberate distinction has been made between *objective* and *subjective* context triggering, whereby the latter refers specifically to an app performing actions based on participants’ subjective, self-reported information. For example, “Ecological Momentary Interventions” (EMIs) can support positive behaviour change through in-the-moment, tailored delivery of prompts or coping strategies to participants, without need for direct researcher

involvement [18]. Further, medication reminders based on participants’ self-reported administration have been perceived as useful in user-centred design studies [11], [19].

Two-Way Feedback: In practical applications of ESM, automated feedback is not a substitute for human contact. Allowing researchers to directly prompt lapsing participants, and participants to report issues to researchers, has been identified as useful for studies in individuals with mental illness, for example [20]. Participatory sensing literature suggests that feedback from researchers could act as a non-monetary incentive mechanism, motivating participants as active contributors to a study [21]. Indeed, employing participatory design as part of an ESM app could enable researchers to immediately address study design issues through direct participant feedback [22].

Preference Tailoring: Just as meta-designers cannot predict the requirements of researchers, researchers cannot entirely predict participants’ needs and characteristics. Participants with hectic schedules support manually tailoring survey prompt times [16], while prompts that are delivered excessively or at inconvenient times are likely to frustrate and result in non-compliance [23]. In an empirical study testing this hypothesis, allowing participants to personalise their sampling times significantly increased their responsiveness to surveys [24]. Other research has also proposed the use of personalised survey schedules to increase compliance [25].

B. Existing EUD-ESM Tools

Responding to the programming barrier faced by psychology researchers, a number of tools exist, both as research projects and proprietary systems, to facilitate ESM development. Table I provides a summary of recent and prominent efforts. We searched the ACM, IEEE and Scopus digital libraries using the terms ‘experience sampling’, ‘ecological momentary’, ‘end-user development’, ‘end-user programming’, and ‘smartphone’ to derive tools in research. Further, given the prevalence of such tools in the commercial domain, a standard Google search was used with the search terms listed above, in an attempt to find proprietary examples. Finally, Conner’s resource on ESM creation tools was used to identify further efforts [26]. The table lists these tools in relation to the four potential ESM features previously identified, which are explored in the following section.

1) **Objective Context:** Tools that enable specification of sampling schedules based on objective context present limited functionality. While the proprietary platforms *LifeData*, *MovisensXS* and *EthicaData* enable the GPS tagging of self-reports, none enable the researcher to trigger based on this location, or indeed other sensors. *AWARE* and *Ohmage* are open-source software platforms that could be programmed to enable sensor triggering, and *EthicaData* provides an API through which developers can create and link their own trigger functionality, but none include this functionality in their available state. Of the platforms that do, *Sensus* is the most diverse in its triggering capabilities, supporting external devices as well as on-device sensors. *PartS* additionally provides a visual interface for specifying objective context triggers.

TABLE I
FEATURES OF MODERN EUD-ESM TOOLS

X:NOT IMPLEMENTED/ ○:POSSIBLE EXTENSION/ ✓:IMPLEMENTED

Platform Name	Objective Trigger	Subjective Trigger	Preference Tailoring	Two-Way Feedback
SurveySignal [28]	✗	✗	✗	✗
LifeData [29]	✗	✗	✗	✗
MovisensXS [30]	✗	✗	✗	✗
PsychLog [31]	✗	✗	✗	✗
EthicaData [32]	○	○	○	✗
PACO [33]	✓	○	○	○
AWARE [34]	○	○	○	○
Ohmage [35]	○	○	○	○
Sensus [36]	✓	○	○	○
PartS [21]	✓	✗	✗	✓

2) *Subjective Context*: None of the tools surveyed allow for subjective context triggering, such as performing actions that are contingent on participants' responses to surveys. Such functionality would be necessary in order to provide tailored intervention feedback to participants. However, self-report data in all reviewed tools cannot be interpreted by the apps themselves; any intervention would need to be initiated by the researcher manually after reviewing participants' data.

3) *Preference Tailoring*: Functionality for tailoring to the characteristics of individual participants has also not been implemented in the tools of Table I. With *ESP*, one of the first examples of electronic ESM, participants used palmtop computers that were manually programmed to account for their waking and sleeping times [27]. This is burdensome for researchers, and inflexible to changes in participants' schedules. *Ohmage* and *PartS* allow participants to set their own reminders but the researcher has no ability to add other customisations.

4) *Two-way Feedback*: Communication functionality is understandably limited in existing tools. Given the requirements of anonymity and consistency inherent in ESM research studies, direct researcher-participant communication has potential ethical implications. Moreover, biased communication delivered to some participants, but not others, could bring the validity of collected data into question. Some tools enable study information messages to be sent to all participants, but with no way for these participants to provide feedback, or to have individual interactions. *PartS*, as a participatory sensing platform, is an exception to this, supporting two-way communication as a motivational incentive for participants.

C. Adoption of EUD in Practice

The prevalence of lab-based usability studies in the evaluation of EUD tools is contrasted by the lack of research into their real-world utility [37], a disparity recognised within HCI as a whole [38]. EUD evaluations are largely focused on the development paradigm and how users' mental models of programming tasks affect the usability of particular paradigms. However, the development of *useful* EUD environments requires knowledge of who potential end-users are, their goals

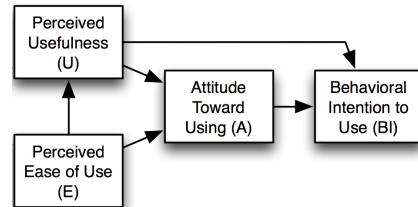


Fig. 2. The Technology Acceptance Model, adapted from [41]

and motivations, and how such environments could fit with current working practices.

As an example of this, recent work by Namoun et al. discusses design implications for mobile EUD from results of surveys and focus groups [39]. The qualitative methods employed, and subsequent data analysis, provide detailed design implications for informing future work. Tetteroo et al. investigate socio-technical factors of introducing EUD in a clinical setting [40], providing deployment guidelines that transcend factors of usability in this domain.

In work outside of EUD, models have been derived that predict the adoption success of general technology, including the Technology Acceptance Model (TAM) [41], illustrated in Figure 2, and the Unified Theory of Acceptance and Use of Technology (UTAUT) [42]. Core factors of both these models are that users' intention to adopt technology is influenced by both its usability, and usefulness. Although these models are applicable to a variety of software, EUD is unusual, in that it involves end-users in a non-traditional role as developers. A question of this research is thus whether these factors are sufficient to capture acceptance of EUD tools in practice.

III. TOOL EXTENSION

For our required evaluations, we updated our existing EUD-ESM tool, *Jeeves*, which employs a visual programming paradigm, specifically a blocks-based approach similar to that of the App Inventor [43] environment. An evaluation of this paradigm showed that non-programmers did not significantly differ in task time or error rate from programmers, and that both perceived *Jeeves* as usable across many dimensions [7]. With an aim to develop *Jeeves* into a tool that supports meta-design, we derived and implemented features to extend and enhance its current functionality, rethinking the desktop interface and smartphone app as "software shaping workshops" as proposed in previous meta-design literature [10].

At the researcher's level, our extended version of *Jeeves* acts as a *system workshop* where researchers create and modify apps to suit their research question or participant group. The app itself acts as an *application workshop* whereby participants can potentially customise their experience to fit their everyday lives. The modular implementation of *Jeeves*, where functionality is composed of different block types, supports simple extension by meta-designers to cope with the changing requirements of researchers. This section discusses our implemented extensions and features, which we performed as part of Figure 1A.

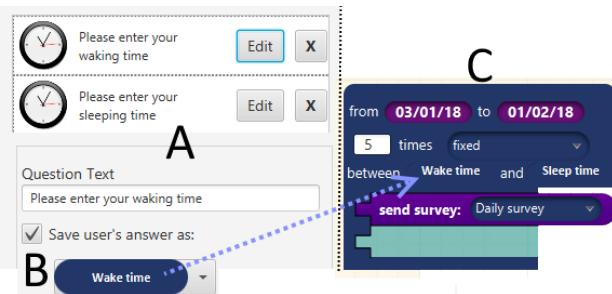


Fig. 3. App behaviour can be customised by storing answers to survey questions in attribute blocks

A. User Data Pane

A new pane was implemented to provide a visual interface to real-time, incoming participant data, and to allow feedback to be sent and received (Figure 1C). This “User Data Pane” consists of a simple GUI where the number of surveys completed and missed by each participant can be viewed. Additionally, **two-way feedback** is afforded through a messaging widget that displays messages sent to and from individual participants via their app.

B. User Attributes

User Attributes were realised as a series of blocks that represent state characteristics of an individual participant, analogous to program instance variables. *Jeeves* contains a new pane for attribute creation, and by using these attributes in a specification, researchers can implement participant-specific app behaviour, as part of their EUD activities (Figure 1B), enabling **subjective context triggering** to be incorporated.

Apps must incorporate participant preferences. For example, participants could select individual waking and sleeping times, or specific locations to receive prompts. In our extension, attribute values are set by survey responses, affording participants **preference tailoring** functionality by answering assigned questions. An example of the process is demonstrated in Figure 3. In this example, the survey is designed to ask participants what their waking and sleeping times are (A). Their answer to the “waking time” question is saved into the “Wake time” attribute (B), which is then used to tailor the schedule of a time-based trigger (C), along with a corresponding “Sleep time” attribute. Participants can then customise a variety of attributes. For example, a survey question can prompt the participant for a GPS location, to customise a geofencing trigger, an example of which is shown in Figure 4.

C. Context-Sensitive Triggering

The previous version of *Jeeves* allows creation of triggering functionality based on a change in location or acceleration. To allow for richer context-sensitive sampling, our updated *JeevesAndroid* smartphone app employs Google’s Activity Recognition API, allowing a variety of activities to be used as triggers. Further, the Google Places API allows geofences to be set at *semantic* locations specified by a participant, such

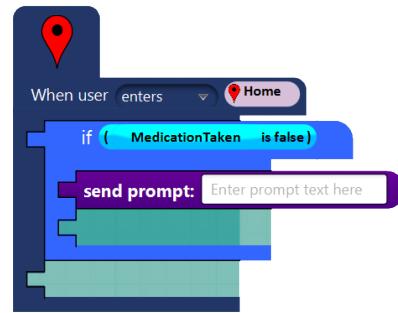


Fig. 4. Objective context triggers can be set up on semantic locations, and combined with other participant states

as their home or workplace. An example of this **objective context triggering** is shown in Figure 4, where a participant’s specified “Home” location, and their current medication state, are combined to trigger a medication reminder prompt.

IV. FACTORS FOR ADOPTION

Following extensions in *Jeeves* with features perceived to be particularly beneficial to ESM studies, we sought to investigate whether these features would be influential in its adoption by social psychology researchers, and further, whether adoption would be contingent on other factors previously unconsidered. Although we separately evaluated the usability of these features, the “perceived ease-of-use” construct of the TAM in Figure 2 is not the focus of this work, as previously discussed. Instead, “perceived usefulness” was evaluated through qualitative research with potential end-users.

A. Interviews

Semi-structured interviews were conducted with five social psychology researchers at our university, recruited through personal email requests. Their research areas and relevant participant cohorts were sufficiently diverse to obtain a range of considerations for adoption of *Jeeves*. Interviews took place at the researchers’ location of choice, lasted approximately 45 minutes, and were organised around three questions:

- **Current practices:** What are researchers’ existing practices in the study of participants, their benefits and drawbacks?
- **Technology use:** What are researchers’ perceptions and experiences of technology in their current practices, and what motivates them to use such technology?
- **Initial impressions:** After being shown an example specification created with *Jeeves*, can researchers envision further applications?

Thematic coding linked researchers’ feedback to factors in the aforementioned existing models of technology acceptance [42]. The relevant factors are *perceived usefulness* (which is further divided into *perceived potential*, *initial requirements* and *participant requirements*) and, exclusive to the UTAUT, *facilitating conditions*.

B. Perceived Potential

The perceived potential benefits of *Jeeves* were elicited from researchers following demonstration of the tool's capabilities. These were primarily related to their difficulty in conducting ecologically valid research, described as follows, and summarised in Table II. These potential benefits are specific to ESM apps, but broad motivations of **saved time**, **functional quality** and **participant quality** are applicable to all public-outward EUD.

Jeeves could enable compliance monitoring

Compliance remains a major issue for studies where participants are required to maintain active participation outside a lab environment, and researchers explained that significant time was wasted through participant dropout. P4 explained how tracking the number of completed surveys, as well as the time taken to complete these surveys, could be used to motivate participants with additional financial compensation:

"if you could have a mechanism I guess of...you know recording if you complete all these bits we'll put you in a prize draw...and you can see if they've done the whole survey in 10 seconds"

The ability to manually prompt participants to complete surveys was also valued by P3, who explained his use of Qualtrics software in order to avoid sending unnecessary compliance emails:

"[Qualtrics] keeps track of who's not responded yet so you can send up a follow-up email to only those who've not responded...allows you to interact with your participant pool"

Jeeves could eliminate recall inaccuracy

Alleviating recall bias and ecological validity issues caused by lab-based data collection was the most significant perceived benefit. Researchers described how a longitudinal experience sampling study would help to alleviate the memory biases that occur due to the time lapse between an event and its reporting, and collect data of a quality previously unattainable.

"The closer you can get to the actual event, and treating each event as a unit...we get a bit closer to the raw experience itself in some way." (P3)

Jeeves could enable capture of contextual information

Context-contingent assessments were perceived as highly desirable in practical applications of *Jeeves*. P5 explained a study she hoped to conduct investigating contextual influences on participants' mindfulness, such that location-based triggers would allow accurate, quality information to be collected:

"That's something we want to develop. 'Where are you? Are you in your bedroom, meditating? Are you outside?' That could be extremely useful"

In situ, repeated self-reports were acknowledged as potentially disruptive. However, researchers suggested that contextual triggers could minimise the number of unnecessary interruptions. P4 described a further potential application of the location trigger functionality to minimise interruptibility:

"If you had the location, you could, as soon as they left...at that point it's appropriate right now to ask them

TABLE II
PERCEIVED POTENTIAL OF JEEVES, FROM INTERVIEW FEEDBACK

Specific Benefit	Broad Motivation	Required Feature(s)
Researchers can prompt compliance	Saved Time	Two-Way Feedback
Researchers acquire in-the-moment data	Functional Quality	
Researchers reduce recall inaccuracy	Functional Quality	Objective Triggering Subjective Triggering
Time saved through remote research	Saved Time	
Participants engage in self-monitoring	Participant Quality	Preference Tailoring
Intervention for sensitive participants	Participant Quality	Subjective Triggering Two-way Feedback

what happened...it's going to be fresh in their mind without interrupting their experience, that'd be great"

Jeeves could save time over traditional data collection

The cross-sectional nature of most research was apparent, and researchers were explicit about the disadvantages of this approach. P4's research of experiences at crowd events required him to visit these events of interest and distribute paper surveys for data collection. Experimental lab research also poses disadvantages, including time and recruitment issues:

"it's difficult to get people into the lab in the first place, to recruit them, it takes a lot of time to organise 'cause you can only do 5 or 6 a day at most, and then once they've done that study they can't really do another one"

Regarding this issue, P2 discussed the benefits of using mobile methods, particularly in gathering data in difficult settings, and eliminating the need to manually transcribe data from paper surveys:

"you can collect data in the field far more easily, collect data in a variety of settings. You don't need a desk to sit and write, you don't need paperwork to collect"

Jeeves could enable self-monitoring

P2 suggested that with intellectually disabled participants, *Jeeves* could be used as a monitoring tool to self-regulate and record dietary habits. The possibility for participants to view and track their data over a period of time, and to receive automatically prompted feedback from an app, could support them to independently manage their health and well-being.

"People with intellectual disabilities are more likely to have various forms of epilepsy because of cognitive damage. So again it would be again your parallel with diabetes, they would be more able to monitor the frequency of seizures and...that would give good feedback for consultants and others"

P2 additionally described how mobile phone use in intellectually disabled populations has dramatically increased in recent years, and that such technology is seen as an asset to their independence.

TABLE III
INITIAL REQUIREMENTS FOR RESEARCHERS

Requirement	Feature Required	Possible Solutions
#1: Researcher Collaboration	Cultures of Participation	Shared editing and semantic annotation
#2: Peer-oriented support	Cultures of Participation	Library of example specs, community forum
#3: Low barrier to entry	Software Shaping Workshops	Domain-specific visual programming paradigm
#4: High ceiling of capabilities	Software Shaping Workshops	Workshops that evolve through feature requests

Jeeves could support sensitive participants

Researchers supported the potential for delivery of feedback to participants, both automated and person-to-person. For example, P2 described working with intellectually disabled individuals, who have a carer or guardian whom they contact for support. Incorporating a means for direct communication between participants and carers was thus considered useful in practical applications of *Jeeves*:

“...if you’d something like this with a button that alerted carers, that’s better, that’s less intrusive than walking about with a band...for self-monitoring, bullying purposes you’ve got some sort of button where they can communicate with people and say ‘I’m not feeling safe’”

As well as enabling direct support from human sources, this could be supplemented with automated support for participants for whom direct researcher contact may not be feasible. P1 endorsed this possibility:

“Wherever we do research where there is a possibility of causing distress we have to take that incredibly seriously...we could automate provision of support to some extent, or at least automate the beginnings of providing support”

Summary

It was promising to observe that the extended features of *Jeeves* were particularly well-received by the interviewed researchers. Each researcher conceived how these features would be conducive to saving time, improving data quality over current methods, and improving participants' overall experience, all of which were considered to be antecedents of adoption. While some advantages were inherent in general smartphone ESM, these were still grounded in the general adoption factors for public-outward EUD.

C. Initial Requirements

The perceived usefulness of a public-outward EUD tool is not only contingent on its theoretical potential benefits, but on its existing functionality that would enable these benefits to be realised. It was noted that these initial requirements were focused less on particular features, but were instead related to concepts in the meta-design approach in Figure 1.

#1: Allow collaboration within/between research groups

In collaboration with peers, or as supervisors to students, the researchers work in teams with varying experience. Two

researchers highlighted a recent “replicability crisis” in science [44], such that research groups developing software may be indirectly developing for future groups to ensure replicability of studies. Both within and between research groups, community support could scaffold ease-of-use [45]. P2 explained how research was typically conducted as part of a team with different specialities:

“You’ve usually got somebody who’s very up on the evidence, very up on the research, but not necessarily technically that competent...then you’ve got somebody else on the team that says right I know how to do [programming]”

Thus, meta-designers for EUD-ESM must consider not only the usability and quality assurance of developed artifacts (public-outward EUD) but also best practices for encouraging collaboration between researchers (public-inward EUD) [8].

#2: Scaffold learning through peer-oriented support

Acceptance of technology is contingent on adequate support for learning and applying its features. P1, in transitioning from SPSS to R statistical software, expressed how such support had enabled her to learn complex functionality. She consults documentation when performing complex tasks, rather than learning how to do these tasks independently:

“By the time I started using it there was that critical mass of people who were developing wikis and stack overflow and this that and the other...I’m not very good at using R but I am good at Googling how to do what I want to do”

As a particular design consideration, an EUD tool should be designed for a spectrum of end-users. At one end are novices, who consult documentation and relevant examples to develop a study fit for their purposes; at the other end are “power users” who explore all features of an interface, and scaffold typical examples that novice end-users could apply.

#3: Allow researchers to perform simple tasks

The various functions in *Jeeves* for tailoring apps to individuals and triggering based on contextual information were perceived as useful to researchers, but could reduce usability by introducing unnecessary complexity to novice users. Researchers valued the idea of pre-created examples with standardised questions, that could then be tailored if necessary. P2 appreciated the ability to tailor based on attributes, but suggested that this functionality could be introduced in time:

“It’s a question of...is there a point at which you need to introduce attributes or do you need to have that there from the start? I don’t know what the answer to that is. You would always start with survey design or blocks, whichever”

Distinct levels of technical self-efficacy arose within the small sample of researchers, suggesting that while complex functionality should be provided, common tasks should be simple and intuitive to ensure that time is saved initially.

#4: Support a high ceiling of functionality

Complementary to the previous requirement, the desire for complex functionality was also expressed by all researchers. P1 expressed how the “low ceiling” of what she could accomplish with SPSS forced her to transition to the more complex R software and begin the learning process again:

"SPSS is very easy to pick up, but you reach a point very quickly where what you want to do is beyond the scope of what it really does and then you have to give up and move to R and start at the bottom of the learning curve again"

Moreover, researchers were averse to software that was inflexible to their diverse needs. P3, a researcher with a technical background, resorted to manual development of generic software to accomplish his goal that purpose-built software did not provide:

"In a sense what I'm doing is press-ganging a more generic piece of software into this kind of mechanism. I mean it can be done, but it's sub-optimal in that sense"

From this perspective, it is important that non-programmer researchers are able to request functionality that meets their needs (Figure 1D), otherwise this functionality will be sought in more complex software, or in a professional programmer.

D. Participant requirements

Further considerations were centred on participants' willingness to comply with a study, highlighting how both researchers and participants have their own separate acceptance criteria. Researchers expressed concerns that an app that is difficult to use, or an app that is intrusive regarding the data it collects, will be quickly removed by participants, and suggested requirements for the smartphone app to improve acceptance.

Assure participants of confidentiality

Researchers explained how conducting remote research requires participants to feel assured that their data is being collected with complete confidentiality. The primary means of doing this is by explicitly giving participants assurances throughout a study:

"...making it very clear to them what was involved in terms of sharing of information...it wouldn't be a sophisticated process but it would need to be something that is done very clearly" (P3)

However, assuring participants of sensitive data storage also relies on an app's reliability in the field, such that there is an explicit need for professional appearance and functionality:

"It needs to look professional, and intuitive...if it's really slow, going between pages, then people are gonna give up. People generally have quite a low threshold I think for some of the stuff" (P4)

Support participants with different skills

Populations with mental and physical disabilities are often ideal participants for social psychology research. It was also acknowledged that many participants would have poor literacy, such that other means of communicating information would need to be employed. P2 suggested graphical depictions of instructions and answers to survey questions:

"Who is it suitable for? If you try and make it too text-heavy you're talking about a fairly narrow group of people but if you open it up with emojis and symbols then you've got something that's more user-friendly"

In summary, with respect to public-outward EUD, the acceptability of both the group performing the development

activities, and the group using the developed artifact, must be taken into consideration separately.

E. Facilitating Conditions

As modelled in the UTAUT (but not in the TAM), **facilitating conditions** also arose, which refer to organisational and technical constraints that could prevent adoption of EUD in work practices regardless of individuals' formed intentions. Fortunately, university research appeared to impose relatively few organisational barriers, crucial to usage behaviour. Indeed, three of the five researchers expressed an interest in using *Jeeves* immediately, two of whom we are currently assisting in their own projects.

The primary facilitating condition expressed by psychology researchers was the affordability of software, mentioned as a key concern by all five researchers:

"Affordability is obviously a big thing so...one of the reasons we were speaking about Qualtrics because not only does it seem to be the market leader but it's also...we have a university licence for that which is a major, a major issue" (P4)

A second factor relates to software already in use, including *Qualtrics*, and statistical software such as *SPSS* and *R*. Researchers were excited about the new possibilities afforded by *Jeeves*, but to minimise integration time, required it to integrate smoothly with current software:

"Inevitably there's gonna be things it can't do, and so being able to actually integrate smoothly...capacity to have that interoperability, plug-in capability, developing sort of thing would be great" (P3)

V. DISCUSSION

The interviews provided a wealth of information on the work practices of social psychology researchers. Having established the perceived usefulness, ease-of-use, and facilitating conditions of *Jeeves*, our discussion relates these findings to the initial question of adoption factors of EUD in this domain. In particular, we note the recurrent themes of **time** and **quality** in determining acceptance, which we propose are integral to general acceptance of public-outward EUD.

Time appears to be the most critical barrier faced by researchers, thus the time *Jeeves* would ultimately save (*perceived usefulness*) the time it would require to learn and use (*perceived ease-of-use*) and the time constraints of particular research projects (*facilitating conditions*) are determining factors for adoption.

However, the time *Jeeves* would save is contingent on the specific goals of researchers, which are not pre-defined. For example, in our ongoing case studies, time-saving qualities (such as a means to obtain informed consent, or collaboration features) emerged through researchers' direct use, and were not previously considered. This implies that a meta-design implementation, where end-users have a stake in design *during* use, is a necessary factor for sustained adoption. When researchers were presented with *Jeeves*, they were able to articulate their time-saving requirements easily in terms of

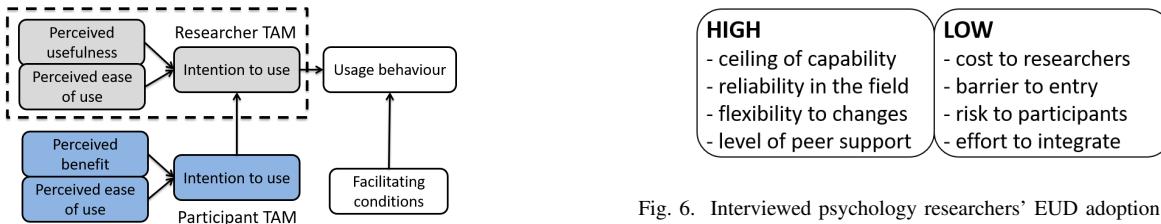


Fig. 5. An informal Technology Acceptance Model of public-outward EUD, requiring benefit and usability to be perceived by all stakeholders

blocks. Such a representation that allows end-users to communicate their requirements effectively (Figure 1D) is conducive to meta-design, and therefore time-saving features.

Quality is another overarching factor discussed. First, the quality of an app in terms of its functionality is a determining adoption factor (*perceived usefulness*), but particularly in terms of its reliability. A reliable app ensures that constant debugging and participant frustration are minimised (*perceived ease-of-use*), but is also necessary to ensure that apps will not cause harm by malfunctioning (*facilitating conditions*).

Functional quality is critical for adopting new ESM technology. Researchers are already comfortable with using *Qualtrics* software, which fulfills their needs with regards to survey creation. Although software that evolves to the needs of its users is key to ensuring that apps are fit-for-purpose, researchers also have initial requirements that must be satisfied by software which, as P4 expressed, “*let us do that which we couldn’t otherwise do*”. While needs vary between end-users, the features derived in Section II, namely: context-sensitivity, participant tailoring, automated feedback, and two-way feedback, were considered desirable by researchers.

A. Public EUD adoption

While researchers have their own model of technology acceptance, this is separate from the acceptance model of their participants. Perceived benefits may overlap, but some are mutually exclusive. Further, initial requirements are also separate, given that researchers and participants interact with two different interfaces. The models are linked, in that researchers will only consider adoption of EUD technology if their participants would be likewise willing to adopt its resultant artifact. Thus, we derived a layered model of technology acceptance, illustrated in Figure 5, to represent public-outward EUD in general.

We also suggest that for adoption of public-outward EUD, it is critical to understand the relationship of domain-experts to their organisation, and to their prospective end-users, prior to engineering. Although we identified two possible applications of ESM, appropriating *Jeeves* (designed as a tool for ESM research) into ESM practice, requires unique considerations. A summary of researchers’ adoption factors, with respect to those that should be high or low, is illustrated in Figure 6.



Fig. 6. Interviewed psychology researchers’ EUD adoption factors

VI. FUTURE WORK & LIMITATIONS

We further describe two additional areas of future work in research goals described in [37], namely in understanding of stakeholders, and the engineering and evaluation of apps created with *Jeeves*.

A. Understanding - Model investigation

While the socio-technical model of ESM development illustrated in Figure 1 was designed with improved acceptance in mind, we did not directly inquire about researchers’ perceptions of specific aspects of the meta-design approach. As part of our ongoing research, we are conducting case studies with researchers to obtain a more in-depth analysis of the utility of these features. Further, we did not attempt to quantify the weight of particular factors in our extended TAM in Figure 5. Future work could probe the true value of these factors in predicting adoption of public EUD in work practices. It is also currently not clear whether this model would generalise to other domains of public-outward EUD.

B. Engineering - Debugging

Application quality and trust are of particular importance when engaging in public-outward EUD. Without knowing how an app will behave prior to its deployment, unexpected functionality issues could critically undermine the utility of ESM. The problem is compounded by the heterogeneity of modern smartphones. Given the privacy concerns surrounding personal data, and the sensitivity of participant groups, researchers must have absolute trust in EUD if they are to adopt it in practice. A suitable testing and debugging framework for researchers could alleviate these issues.

VII. CONCLUSION

The introduction of EUD into professional work practices presents challenges beyond ease-of-use. While the gap between users’ software requirements and their programming capabilities appears suitable to bridge with an EUD tool, it is important to ask why the gap exists, and how we as computer scientists are best placed to fill it. Qualitative research with potential end-users can inform us of the likelihood of an EUD tool’s success, and indeed feedback from interviews has been invaluable in disrupting our assumptions of what ESM apps could or should do for psychology researchers. Our goal in extending *Jeeves* was not to simply append new features, but to develop it into a system that would allow useful features to be proposed and incorporated as required by end-users. Acceptance of EUD technology is a dynamic process that will require continuous feedback from all stakeholders.

REFERENCES

- [1] R. Larson and M. Csikszentmihalyi, "The experience sampling method," in *Flow and the foundations of positive psychology*. Springer, 2014, pp. 21–34.
- [2] S. Shiffman, A. A. Stone, and M. R. Hufford, "Ecological momentary assessment," *Annual Review of Clinical Psychology*, vol. 4, pp. 1–32, 2008.
- [3] B. Lenaert, M. Colombi, C. van Heugten, S. Rasquin, Z. Kasanova, and R. Ponds, "Exploring the feasibility and usability of the experience sampling method to examine the daily lives of patients with acquired brain injury," *Neuropsychological Rehabilitation*, pp. 1–13, 2017.
- [4] V. Pejovic, N. Lathia, C. Mascolo, and M. Musolesi, "Mobile-based experience sampling for behaviour research," in *Emotions and Personality in Personalized Services*. Springer, 2016, pp. 141–161.
- [5] N. V. Berkel, D. Ferreira, and V. Kostakos, "The experience sampling method on mobile devices," *ACM Computing Surveys (CSUR)*, vol. 50, no. 6, p. 93, 2017.
- [6] H. Lieberman, F. Paternò, M. Klann, and V. Wulf, "End-user development: an emerging paradigm," in *End User Development*. Springer, 2006, pp. 1–8.
- [7] D. Rough and A. Quigley, "Jeeves-a visual programming environment for mobile experience sampling," in *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. IEEE, 2015, pp. 121–129.
- [8] F. Cabitzka, D. Fogli, and A. Piccinno, "'Each to his own': distinguishing activities, roles and artifacts in EUD practices," in *Smart Organizations and Smart Artifacts*. Springer, 2014, pp. 193–205.
- [9] A. Mørch, "Three levels of end-user tailoring: customization, integration, and extension," pp. 51–76, Nov 1997.
- [10] F. Paternò and V. Wulf, *New Perspectives in End-User Development*. Springer, 2017.
- [11] L. Simons, A. Z. Valentine, C. J. Falconer, M. Groom, D. Daley, M. P. Craven, Z. Young, C. Hall, and C. Hollis, "Developing mhealth remote monitoring technology for attention deficit hyperactivity disorder: a qualitative study eliciting user priorities and needs," *JMIR mHealth and uHealth*, vol. 4, no. 1, 2016.
- [12] J. Os, S. Verhagen, A. Marsman, F. Peeters, M. Bak, M. Marcelis, M. Drukker, U. Reininghaus, N. Jacobs, T. Lataster *et al.*, "The experience sampling method as an mhealth tool to support self-monitoring, self-insight, and personalized health care in clinical practice," *Depression and anxiety*, vol. 34, no. 6, pp. 481–493, 2017.
- [13] P. Klasnja and W. Pratt, "Healthcare in the pocket: mapping the space of mobile-phone health interventions," *Journal of Biomedical Informatics*, vol. 45, no. 1, pp. 184–198, 2012.
- [14] V. Pejovic and M. Musolesi, "InterruptMe: designing intelligent prompting mechanisms for pervasive applications," in *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, 2014, pp. 897–908.
- [15] M. N. Burns, M. Begale, J. Duffecy, D. Gergle, C. J. Karr, E. Giangrande, and D. C. Mohr, "Harnessing context sensing to develop a mobile intervention for depression," *Journal of medical Internet research*, vol. 13, no. 3, 2011.
- [16] N. Ramanathan, D. Swendeman, W. S. Comulada, D. Estrin, and M. J. Rotheram-Borus, "Identifying preferences for mobile health applications for self-monitoring and self-management: focus group findings from HIV-positive persons and young mothers," *International journal of medical informatics*, vol. 82, no. 4, pp. e38–e46, 2013.
- [17] J. D. Runyan, T. A. Steenbergh, C. Bainbridge, D. A. Daugherty, L. Oke, and B. N. Fry, "A smartphone ecological momentary assessment/intervention "app" for collecting real-time data and promoting self-awareness," *PLoS One*, vol. 8, no. 8, 2013.
- [18] K. E. Heron and J. M. Smyth, "Ecological momentary interventions: incorporating mobile technology into psychosocial and health behaviour treatments," *British Journal of Health Psychology*, vol. 15, no. 1, pp. 1–39, 2010.
- [19] M. E. Hilliard, A. Hahn, A. K. Ridge, M. N. Eakin, and K. A. Riekert, "User preferences and design recommendations for an mhealth app to promote cystic fibrosis self-management," *JMIR mHealth and uHealth*, vol. 2, no. 4, 2014.
- [20] J. E. Palmier-Claus, I. Myint-Germeys, E. Barkus, L. Bentley, A. Udachina, P. Delespaul, S. W. Lewis, and G. Dunn, "Experience sampling research in individuals with mental illness: reflections and guidance," *Acta Psychiatrica Scandinavica*, vol. 123, no. 1, 2011.
- [21] T. Ludwig, J. Dax, V. Pipek, and D. Randall, "Work or leisure? Designing a user-centered approach for researching activity "in the wild"," *Personal and Ubiquitous Computing*, vol. 20, no. 4, pp. 487–515, 2016.
- [22] K. Shilton, N. Ramanathan, S. Reddy, V. Samanta, J. Burke, D. Estrin, M. Hansen, and M. Srivastava, "Participatory design of sensing networks: strengths and challenges," in *Proceedings of the Tenth Anniversary Conference on Participatory Design 2008*. Indiana University, 2008, pp. 282–285.
- [23] L. Dennison, L. Morrison, G. Conway, and L. Yardley, "Opportunities and challenges for smartphone applications in supporting health behavior change: qualitative study," *Journal of medical Internet research*, vol. 15, no. 4, 2013.
- [24] P. Markopoulos, N. Batalas, and A. Timmermans, "On the use of personalization to enhance compliance in experience sampling," in *Proceedings of the European Conference on Cognitive Ergonomics 2015*. ACM, 2015, p. 15.
- [25] S. Vhaduri and C. Poellabauer, "Human factors in the design of longitudinal smartphone-based wellness surveys," in *Healthcare Informatics, 2016 IEEE International Conference on*. IEEE, 2016, pp. 156–167.
- [26] T. S. Conner, "Experience Sampling and Ecological Momentary Assessment with Mobile Phones," 2015.
- [27] L. F. Barrett and D. J. Barrett, "An introduction to computerized experience sampling in psychology," *Social Science Computer Review*, vol. 19, no. 2, pp. 175–185, 2001.
- [28] <http://www.surveysignal.com>, accessed: 3rd April 2018.
- [29] <http://www.lifedatacorp.com>, accessed: 3rd April 2018.
- [30] <http://xs.movisens.com>, accessed: 3rd April 2018.
- [31] A. Gaggioli, G. Pioggia, G. Tartarisco, G. Baldus, D. Corda, P. Cipresso, and G. Riva, "PsychLog: A mobile data collection platform for mental health research," *Personal and Ubiquitous Computing*, vol. 17, no. 2, pp. 241–251, Feb 2013.
- [32] <http://www.ethicadata.com>, accessed: 3rd April 2018.
- [33] <http://www.pacoapp.com>, accessed: 3rd April 2018.
- [34] D. Ferreira, V. Kostakos, and A. K. Dey, "AWARE: mobile context instrumentation framework," *Frontiers in ICT*, vol. 2, p. 6, 2015.
- [35] H. Tangmunarunkit, C.-K. Hsieh, B. Longstaff, S. Nolen, J. Jenkins, C. Ketcham, J. Selsky, F. Alquaddoomi, D. George, J. Kang *et al.*, "Ohmage: a general and extensible end-to-end participatory sensing platform," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 6, no. 3, p. 38, 2015.
- [36] H. Xiong, Y. Huang, L. E. Barnes, and M. S. Gerber, "Sensus: a cross-platform, general-purpose system for mobile crowdsensing in human-subject studies," in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, 2016, pp. 415–426.
- [37] D. Tetteroo and P. Markopoulos, "A review of research methods in end user development," in *International Symposium on End User Development*. Springer, 2015, pp. 58–75.
- [38] S. Greenberg and B. Buxton, "Usability evaluation considered harmful (some of the time)," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2008, pp. 111–120.
- [39] A. Namoun, A. Daskalopoulou, N. Mehandjiev, and Z. Xun, "Exploring mobile end user development: existing use and design factors," *IEEE Transactions on Software Engineering*, vol. 42, no. 10, 2016.
- [40] D. Tetteroo, P. Vreugdenhil, I. Grisel, M. Michielsen, E. Kuppens, D. Vanmulken, and P. Markopoulos, "Lessons learnt from deploying an end-user development platform for physical rehabilitation," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM, 2015, pp. 4133–4142.
- [41] V. Venkatesh and F. D. Davis, "A theoretical extension of the technology acceptance model: four longitudinal field studies," *Management science*, vol. 46, no. 2, pp. 186–204, 2000.
- [42] V. Venkatesh, M. G. Morris, G. B. Davis, and F. D. Davis, "User acceptance of information technology: toward a unified view," *MIS quarterly*, pp. 425–478, 2003.
- [43] F. Turbak, D. Wolber, and P. Medlock-Walton, "The design of naming features in App Inventor 2," in *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*. IEEE, 2014, pp. 129–132.
- [44] M. Baker, "1,500 scientists lift the lid on reproducibility," *Nature News*, vol. 533, no. 7604, p. 452, 2016.
- [45] M. Spahn, C. Dörner, and V. Wulf, "End user development: approaches towards a flexible software design," in *ECIS*, 2008, pp. 303–314.

Calculation View: multiple-representation editing in spreadsheets

Advait Sarkar*, Andrew D. Gordon*†, Simon Peyton Jones*, Neil Toronto*

*Microsoft Research, 21 Station Road, Cambridge, United Kingdom

†University of Edinburgh School of Informatics, 10 Crichton Street, Edinburgh, United Kingdom
 {advait,adg,simonpj,netoront}@microsoft.com

Abstract—Spreadsheet errors are ubiquitous and costly, an unfortunate combination that is well-reported. A large class of these errors can be attributed to the inability to clearly see the underlying computational structure, as well as poor support for abstraction (encapsulation, re-use, etc). In this paper we propose a novel solution: a *multiple-representation* spreadsheet containing additional representations that allow abstract operations, without altering the conventional grid representation or its formula syntax. Through a user study, we demonstrate that the use of multiple representations can significantly improve user performance when performing spreadsheet authoring and debugging tasks. We close with a discussion of design implications and outline future directions for this line of inquiry.

I. INTRODUCTION

Spreadsheets excel at showing data, while hiding computation. In many ways the emphasis on showing data is a huge advantage, but it comes with serious difficulties: because the computations are hidden, spreadsheets are hard to understand, explain, debug, audit, and maintain.

It is often remarked that “spreadsheets are code” [1]. What would happen if we take that idea seriously, and offer a view of the spreadsheet designed primarily to display its computational structure? Then, in this *Calculation View*, we might be able to offer more abstract operations on ranges within the grid, and alternative ways to achieve useful tasks that are cumbersome or error-prone in the grid view. We have designed, prototyped, and evaluated just such a feature (Fig. 1). More specifically, we make the following contributions.

- We present a design for a view of a spreadsheet primarily intended for viewing formulas and their groupings. Edits to either the grid or to Calculation View show up immediately in the other. This design and its possible variants are discussed in the context of the theory of multiple representations (Sections III and IV).
- We describe two particularly compelling advantages of Calculation View:
 - Calculation View improves on error-prone copy/paste (Section III-B) using *range assignment*: a new textual syntax for copying a formula into a block of cells.
 - Calculation View offers a simple syntax for naming cells or ranges, and referring to those names in other formulas (Section III-C). Naming is available in spreadsheets such as Excel, but few users exploit it because of the high interaction cost.

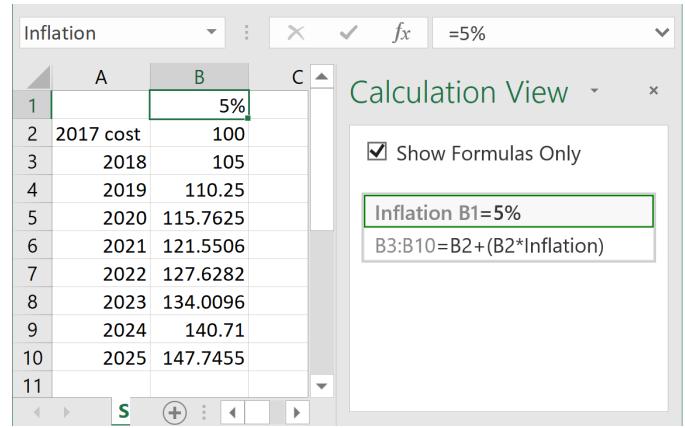


Fig. 1. Calculation View lists the formulas in a spreadsheet. It enables abstract operations such as range assignment and cell naming.

- We present the results of a user study (Section V) showing that certain common classes of spreadsheet authoring and debugging tasks are faster when users have access to Calculation View, with lower cognitive load, without reduction in self-efficacy.

We regard Calculation View as a first step in a rich space of multiple-representation designs that can enable new experiences in spreadsheets, discussed further in Section VI.

II. THE PROBLEM AND OUR APPROACH

A. Problem: errors in spreadsheets

As with any large body of code, spreadsheets contain errors of many kinds, with often catastrophic implications, given the heavy dependence on spreadsheets in many domains. The ubiquity and maleficence of spreadsheet errors has been well documented [2], and there are even specialised conferences dedicated solely to spreadsheet errors!¹

We focus on the following specific difficulties, using the vocabulary of cognitive dimensions [3]:

- 1) *Invisibility of computational structure*. The graphical display of the sheet does not intrinsically convey how values are computed, which groups of cells have shared formulas, and how cells depend on each other. This creates *hidden dependencies* in the sheet’s dataflow.

¹<http://www.eusprig.org/>

Apart from individually inspecting cell formulas, or relying on secondary notation provided by the spreadsheet author (layout, borders, whitespace, colouring, etc.), there are no affordances for auditing the calculations of a spreadsheet, which makes auditing tedious and error-prone. *Visibility* suffers in large spreadsheets; the display is typically too small to contain all formulas at once. Visibility is also impaired by the inability to display formulas and their results simultaneously; the user must inspect formulas individually using the formula bar. The “Show formulas” option, which displays each cell’s formula in the cell instead of the computed value, is also impractical, since the length of formulas typically exceeds the cell width, leading to truncation.

- 2) *Poor support for abstraction.* Consider the following common form of spreadsheet:

Data	Formula 1	Formula 2	...	Formula k
d_1	$F_1(d_1)$	$F_2(d_1)$...	$F_k(d_1)$
d_2	$F_1(d_2)$	$F_2(d_2)$...	$F_k(d_2)$
...
d_n	$F_1(d_n)$	$F_2(d_n)$...	$F_k(d_n)$

The first column is a list of data, and each other column simply computes something from the base data. The formulas in each row repeat the calculation for the data in that row; rows are independent. There are only as many distinct formulas as there are columns; the complexity of building and testing this spreadsheet should not be affected by whether there are ten rows, or ten million. The user experience, unfortunately, is deeply affected. The notation is *error prone* in that the user is responsible for manually ensuring that the column formula is precisely copied the correct number of rows. Any subsequent edits to column formulas are *viscous* as well as *error prone*, as they must be correctly propagated to the correct range, which involves identifying all the cells that the author intended to contain that formula, an intention for which there is usually no explicit record.

- 3) *Formulas suffer from a lack of readable names.* Grid cell references (e.g., A1, B2, etc.) are terrible variable names, as they contain no information regarding what the value in the cell might represent. They can be easily mistyped as other valid grid cell references, leading to a silent error. Conventional programming languages allow users to give domain-relevant names to their values (improving *closeness-of-mapping*); for example, we might want to refer to cell B2 as TaxRate – a simple form of abstraction. Some spreadsheet packages do in fact support naming cells and cell ranges (e.g., Excel’s name manager²) but these features are not widely used due to high additional interaction and cognitive costs: of naming cells; of recalling what cells have been named; and remembering to actually use the name (i.e., not mixing usage of the name and the cell it refers to).

B. Our approach: augmenting the grid

Previous approaches to mitigating errors in spreadsheets have focused either on auditing tools, or on modifying the grid and its formula syntax (see Section VII). In this paper, we present an exploration of a fundamentally new approach to the problem. We propose that the grid, and its formula syntax, be left *untouched*, but to provide opportunities for abstraction through *additional representations*. We build on the theory of *multiple representations* that originates in Ainsworth’s research in mathematics education [4] but has found widespread applications in computer science education [5], [6], and end-user programming research [7]. By offering multiple representations of the same core object (in our case, the program exemplified by the spreadsheet), we can help the user learn to move fluently between different levels of abstraction, choosing the abstraction appropriate for the task at hand.

III. TEXTUAL NOTATION IN CALCULATION VIEW

Thus motivated, we created an alternative representation, *Calculation View*, or CV for short, of the spreadsheet as a textual program. CV is displayed in a pane adjacent to the grid. In CV, the grid is described as a set of formula assignments. For example:

B1 = SQRT(A1)

assigns the formula =SQRT(A1) to the cell B1. Edits in one view are immediately propagated to the other and the spreadsheet is recalculated; it is *live* [8].

A. Review: formula copy-and-paste in spreadsheets

Before we introduce a new, more powerful type of assignment in CV, it is helpful to review the distinctive behaviour of copy-and-paste in spreadsheets today.

Suppose that cells A1 to A10 contain some numbers, and the user wishes to compute the square root of each of these numbers in column B. The user would begin by typing =SQRT(A1) into cell B1. They could type =SQRT(A2) into cell B2, and so on, but a more efficient method is to copy =SQRT(A1) from cell B1 and paste into B2. The user intention is not to paste the same literal formula, but rather one that is updated to point to the corresponding cell in A. The operation of *formula copy-and-paste* rewrites the formula =SQRT(A1) into the intended form =SQRT(A2).

This is achieved by interpreting references in the original formula as spatially relative to the cell, as can be expressed using “R1C1” notation. For example, the expression SQRT(A1) occurring in cell B1 is represented as SQRT(R[0]C[-1]) in R1C1, because with respect to B1, A1 represents the cell in the same row (R[0]) and the previous column (C[-1]). This formula pasted into the cells B2 to B10 becomes the sequence SQRT(A2), ..., SQRT(A10); the relative reference resolves into a different cell reference for each case. Spreadsheet packages generally allow this behaviour to be overridden (e.g., Excel’s *absolute references*³).

²<https://support.office.com/en-ie/article/Define-and-use-names-in-formulas-4d0f13ac-53b7-422e-afd2-abd7ff379c64>

³<https://support.office.com/en-us/article/switch-between-relative-absolute-and-mixed-references-dfec08cd-ae65-4f56-839e-5f0d8d0bacaa>

The *drag-fill* operation builds on formula copy-and-paste. In a drag-fill, the user types =SQRT(A1) into cell B1, selects it, and then drags down to cover the range B1:B10, which is equivalent to copying B1 into each cell in the range.

Copy/paste and drag-fill enable the user to create computations on arrays and matrices without needing to understand functional programming formalisms such as `map`, `fold`, and `scan`. However, the conceptual abstraction of arrays is not reflected in any grid affordances; it is easy to accidentally omit cells or overextend the drag-filling operation, and the user must manually propagate any changes in the formula to all participating cells – a fiddly and error-prone process.

CV, being separate from the grid, presents an opportunity to allow abstract operations on arrays and matrices *without* affecting the usability of the grid.

B. First idea: range assignments

The first novel affordance of our notation is *range assignment*, which assigns the same formula to a range of cells just as a drag-fill copies a single formula to a range. In CV, the user could accomplish the previous example using the following range assignment:

B1:B10 = SQRT(A1)

The colon symbol is already used in Excel to denote a range, and so its use capitalises on users' existing syntax vocabulary.

The assignment has an effect identical to entering the formula =SQRT(A1) into the top-left cell of the range B1:B10, and then drag-filling over the rest of the range. Observe how our syntax uses the literal formula for the top-left cell; users must apply their mental model of formula copy-and-paste to predict how the formula will behave for the rest of the range. In this manner, range assignment exposes a low-abstraction syntax for array/matrix assignment.

An alternative, that does not rely on knowledge of copy-paste semantics, would be to use R1C1 notation:

B1:B10 = SQRT(R[0]C[-1])

This is clearer, because the same formula is assigned to every cell, but understanding the formula requires knowledge of the more abstract R1C1 notation.

Range assignment has many benefits. It is less *diffuse/verbose*, as it represents all formulas in a block using a single formula. It has a greater *closeness-of-mapping* to user intent. It greatly improves *visibility* of the formulas in the sheet (take for example our sheet with one formula per column – even with thousands of rows, the CV representation shows a single range assignment per column). Moreover, the representation greatly reduces the *viscosity* and *error-proneness* of editing a block of formulas. Instead of manual copying or drag-filling, the user simply edits the formula in the range assignment. The range itself can also be edited to adjust the extent of the copied formula precisely and easily.

Cells and Ranges:

Cell ::= A1-notation
Range ::= Cell Cell:Cell

Formulas:

Literal ::= number string
Name ::= identifier
Fun ::= SUM SQRT ...
Formula ::=
Literal Range Name Fun(Formula ₁ , ..., Formula _N) ...

Assignments and Programs:

Assignment ::=
Range = Formula
Name Range = Formula
Program ::= Assignment ₁ ... Assignment _N

Fig. 2. Abstract Syntax for Calculation View

C. Second idea: cell naming

The lack of meaningful names for grid cell references leads to unreadability and error proneness in formulas. Extant naming features in spreadsheet packages are seldom used in practice; CV presents an opportunity to drastically lower the interactional and cognitive costs for using names. To name a cell or range, the user employs the following syntax:

Name Cell = Formula

A concrete example is this:

TaxRate A1 = 0.01

which puts the value 0.01 into cell A1 and gives it the name **TaxRate**. Thus to compute tax, one can write the formula in terms of **TaxRate** rather than A1, which is more readable, more memorable, more intelligible, and more difficult to mistype as a different but valid reference. We considered alternative naming syntaxes (e.g., **TaxRate[A1] = ...**; **TaxRate in A1 = ...**; **A1 as TaxRate = ...**; **TaxRate = A1 = ...**; etc.) and a detailed investigation of this would make for interesting future work, but within the scope of our initial exploration we settled on the simple space-delimited syntax for its readability.

D. Summary syntax and semantics for Calculation View

Figure 2 shows the complete grammar of the textual notation in our initial implementation of Calculation View.

Our language has a simple semantics, as follows. An assignment **Range = Formula** is equivalent to entering =Formula into the top-left cell of **Range**, and pasting that formula to every other cell in **Range**. An assignment **Name Range = Formula** additionally binds the name **Name** to the range **Range**.

We require that no two assignments target the same cell. We place other constraints on the program including that each range targets a non-empty set of cells.

IV. INTERACTION DESIGN IN CALCULATION VIEW

A. Use of multiple representations

“Multiple representations” is a broad umbrella term for systems that show some shared concept in multiple ways, but this can have a variety of different manifestations, depending on how tightly coupled the representations are, what underlying concepts they share, and other design variables. CV’s specific use of multiple representations – in particular, what functions CV does, and does not perform – can be characterised in terms of Ainsworth’s functional taxonomy for multiple-representation environments [4]:

- *Complementary roles through complementary tasks.* In CV, these tasks are: creating and editing formulas, creating and editing ranges of shared formulae, and viewing the computational structure of the sheet. In the grid view, these tasks are: setting cell formatting, layout, and other secondary notation to prepare the data for display, inserting charts and other non-formula entities, etc. CV and the grid facilitate *complementary strategies*; the primary strategy for range editing in the grid is copy/paste or drag-fill, which is well suited for small ranges and for visual display of data. In CV, the primary strategy is to use range assignment, which is well suited for robust editing of ranges with shared formulas.
- *Complementary information:* CV can display formulas while the grid displays data and formula output.
- CV constructs deeper understanding using abstraction through reification: a type of abstraction where a process at one level is reconceived as an object at a higher level [9]. In spreadsheets, users understand a range of shared formulas as a single abstract entity; the process of copy/paste or drag-filling at the cell level creates an object at the range level. In CV, we build on that understanding and reify those ranges as single objects.

Our model of shared representation is depicted in Figure 3. Both CV and the grid share certain features, such as the ability to assign names, and the ability to assign formulas to individual cells. However, CV allows range assignment and a naming syntax not possible in the standard grid. Similarly, CV does not have facilities for adjusting cell formatting, or viewing the spatial grid layout of formulas.

CV introduces no new information content to the spreadsheet; indeed, the CV is generated each time the spreadsheet is opened, or when the grid view is edited (see Section IV-C).

B. Editing experience design

CV departs from traditional text editors in a few deliberate ways. The first is the explicit visual distinction between lines, creating a columnar grid of *pseudocells*. This makes CV appear familiar, due to its similarity to the grid, and reinforces the fact that there should only be one assignment per line. Unlike many other programming languages, which permit multiple statements on a single line (delimited by, e.g., semicolons), Excel has no counterpart to this and so CV’s pseudocells help indicate the absence of that facility.

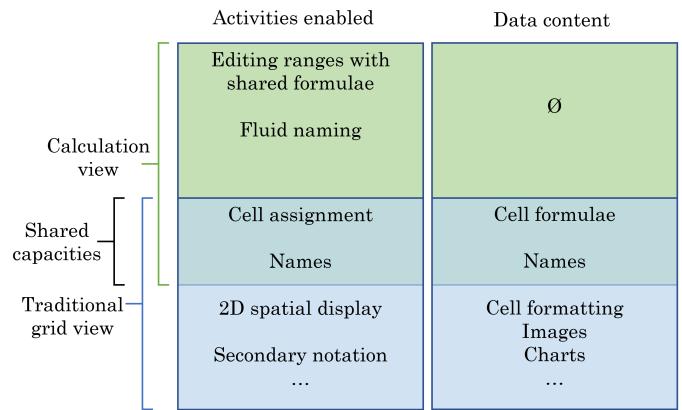


Fig. 3. Relationship between Calculation View and the traditional grid.

The second departure of CV from a simple text editor is the newline behaviour. In the Excel grid, hitting the enter (or return) key has the effect of committing the current formula and moving focus to the next cell down. If this same behaviour were adopted wholesale into CV, then hitting enter would only navigate between pseudocells, and additional interface components would be required to allow users to create *new* cell/range assignments. Instead, in our design, hitting enter while any pseudocell is in focus creates a new pseudocell underneath it, combining the properties of a flat text editor and the grid. Pseudocells can only be empty while they are being edited. If a pseudocell is empty when it loses focus, it disappears. Thus, cell and range assignments can be deleted by deleting the contents of the corresponding pseudocell, and when the pseudocell loses focus, it disappears from CV and so do its formulas on the grid. Another aspect of this design is that unlike in a text editor, where multiple blank lines can be entered by repeatedly hitting enter, in CV repeatedly hitting enter does nothing after the initial empty pseudocell is created – no new pseudocells will be created while an empty pseudocell is in focus.

We acknowledge, however, that the free addition of whitespace and re-ordering of statements is a valuable form of secondary notation in textual programming languages. In future work it would be useful to compare a version of CV presented as a simple text editor, with the pseudocell representation we have created (Section VI).

C. Block detection algorithm

It is not sufficient for CV to *only* display range assignments created in the CV editor. In order to fully capitalise on the increased abstraction possible in CV, *any* block of copied/drag-filled formulas, even if these operations were performed manually in grid view, should also be represented in CV as a range assignment. We implemented a simple block detection algorithm to achieve this. The algorithm operates as follows: the cells in the sheet are first placed into R1C1 equivalence classes (i.e., cells with the same formula in R1C1 are grouped into the same class). Then, for each class, maximal rectangular ranges (called ‘blocks’) are detected using a greedy flood-

filling operation: the top-left cell in the class is chosen to ‘seed’ the block. The cell to the right of the seed is checked; if it belongs to the same class, then the block is grown to include it. This is repeated until the block has achieved a maximal left-right extent. The block is now grown vertically by checking if the corresponding cells in the row below are also part of the equivalence class. Once it can no longer be grown vertically, this maximal block is then ‘removed’ from the equivalence class. A new top-left seed is picked and grown, and the process is repeated until all the cells in the equivalence class have been assimilated as part of a block.

Each block so detected becomes a range assignment in CV. There are edge cases in which the behaviour of our algorithm is somewhat arbitrary. For instance, in an L-shaped region of cells containing R1C1-equivalent formulas, the ‘corner’ of this region could reasonably belong to either ‘arm’, but our greedy approach gives preference to the top-leftmost arm. Blocks of this shape are unusual in practice, and for our initial exploration, our basic approach has proven adequate.

D. Formula ordering

In what order should formulas be listed in CV? There are at least two straightforward options: (1) ordering by cell position (e.g., left to right, top to bottom) and (2) ordering by a topological sort of the formula dependency graph. Both options are viable: the former juxtaposes cells that are spatially related to each other, the latter juxtaposes cells that are logically related. In our investigation we have not addressed this design choice. For simplicity we adopted spatial ordering, but it may be better to allow the user to choose, or to choose using a heuristic characterisation of the spreadsheet.

The user can enter a newline in any pseudocell in CV to create a new pseudocell below it. The formula in this pseudocell can pertain to any cell or range in the grid, and will remain in the position it was entered until another cell in the grid (not CV) is selected, which triggers a regeneration of CV, at which point the formula is moved to its position according to spatial ordering. This is illustrated in Figure 4.

Alternative designs are possible. For instance, the interface might make an exception for formulas entered in CV, remember their position relative to other formulas, and try to preserve that position as well as possible in order to prevent the jarring user experience of having their formula moved around. The problem of preserving position is nontrivial, and would make for interesting future work.

E. View filtering

Even after block detection has collapsed blocks of formulas into single pseudocells, there is still potential for CV to become cluttered. For instance, in the example from Section II, if all the cells containing base data in the first column were displayed in CV, hundreds of pseudocells displaying base data would obscure the range assignments for the other columns – which are the main items of interest. To improve this, CV filters out literal values by default (with the option to show them if necessary). In future work, one might imagine

advanced sorting and filtering functionality, such as “show only formulas within a certain range”, or “show only formulas containing some subexpression”, or “show formulas which evaluate to a certain type, e.g., boolean”, or even simpler options such as “sort by formula length”.

The key observation with respect to view filtering in CV is that, as in other multiple representation systems, each individual representation is suitable/superior for certain specific things. Here, the grid is a *superb* place to display lots of literal values; CV need not compete with the grid for doing that. CV is good at showing formulas and their abstract grouping, so it should have affordances for doing that well.

V. USER STUDY

CV aims to present a higher level of abstraction in spreadsheets without affecting the fundamental usability of the grid. We are interested in whether access to such a representation helps users create and reason about spreadsheets with less manual and cognitive effort.

We refined our research interests into the following concrete hypotheses. Does the addition of CV to the grid affect:

- 1) the time taken to author spreadsheets;
- 2) the time taken to debug spreadsheets;
- 3) the user self-efficacy in spreadsheet manipulation; and
- 4) the cognitive load for spreadsheet usage?

We are also interested in whether any observed difference is affected by the participant’s level of spreadsheet expertise.

A. Participants

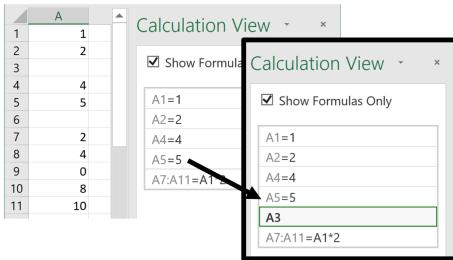
We recruited 22 participants, between 25 and 45 years of age, 14 female and 8 male, using convenience sampling. Participants spanned four different organisations and worked in a range of professions including office administration, real estate planning and surveying, interaction design research, and civil engineering. All 22 had prior experience with spreadsheets and 18 used spreadsheets in regular work.

B. Tasks

We used two types of tasks: authoring and debugging. For the authoring tasks, participants were given a partially completed spreadsheet and asked to complete it. For each authoring task, completion involved writing between 1-3 simple formulas, and copying those formulas to fill certain ranges. We created 2 pairs of authoring tasks, where tasks within a pair were designed to be of equal difficulty. For instance, one task was for participants to calculate several years of appreciated prices for a list of real estate properties whose current values were given. The matched counterpart for this task was for participants to calculate several years of depreciated values for a list of company assets whose current values were given. Both require writing a formula of similar complexity and filling it to a range of similar size.

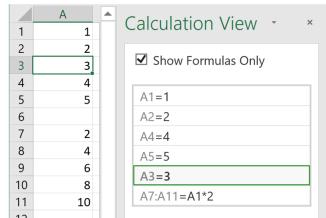
In debugging tasks, participants were given a completed spreadsheet and informed that there may be any of two types of errors: a copy/paste or drag-fill error where a row or column had been accidentally omitted or included, and a cell where

The user positions the cursor at the end of the line 'A5=5' and hits enter, which creates a new line.



The user starts typing an assignment to cell A3

The assignment to cell A3 stays in place while the cursor focus is in Calculation View



When the user clicks elsewhere, Calculation View is re-ordered

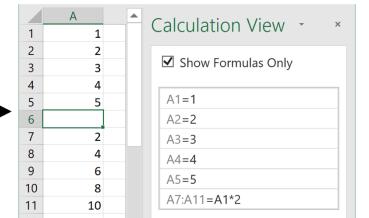


Fig. 4. The user can create assignments at any position in CV. When CV loses focus, assignments are re-ordered according to their spatial ordering.

a formula had been inadvertently overwritten using a fixed constant. The task was to detect any errors of these two types. We created 2 pairs of debugging tasks with matched difficulty. These tasks resembled the completed sheets that the participants were to create in the authoring task, so that the participant already understood what the purpose of the sheet was. In each task there was exactly one drag-fill error and one overwriting error, but participants were not informed of this.

C. Protocol

Participants were briefed and signed a consent form. They then completed a questionnaire about their spreadsheet and programming expertise, based on a questionnaire used in a previous study of program comprehension [10], but refactored to include items specific to spreadsheets. They were then given a 10-minute tutorial covering formulas and drag-filling in the standard public release of Microsoft Excel, as well as the range assignment syntax in CV, and given the opportunity to clarify their understanding with the experimenter.

Participants then completed four tasks: two authoring and two debugging tasks. Half the participants used Excel without CV and the other half used Excel with CV. After these tasks, participants completed standard questionnaires for cognitive load (NASA TLX [11]) and computer self-efficacy [12]. Participants completed a further four tasks, these being the matched counterparts to the tasks in the first round, this time with CV if they were without CV for the first round, or vice versa. After these tasks, participants again completed cognitive load and self-efficacy questionnaires.

The order in which participants encountered our experimental conditions (with or without CV) was balanced, and we could make a within-subjects comparison. The order in which tasks of each type were presented was counterbalanced. Within each task-pair, each task of the pair was assigned alternately to the with-CV and the without-CV condition.

The experiment lasted 70 minutes on average and participants were compensated £20 for their time.

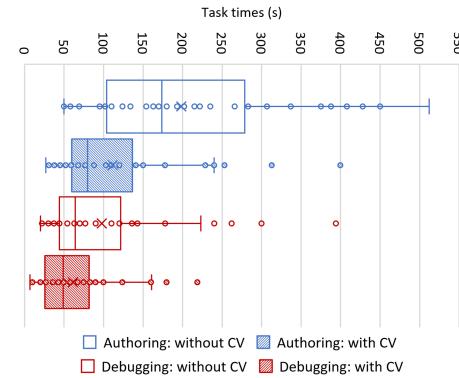


Fig. 5. Task times with and without CV.

D. Results

Task times: Participants took less time to complete spreadsheet *authoring* tasks when using CV than without (median difference of -54 seconds, or a median speed-up of 37.14%). This difference is statistically significant (Wilcoxon signed rank test: $Z = -4.14, p = 3.6 \cdot 10^{-5}$). See Figure 5.

Participants took less time to complete spreadsheet *debugging* tasks when using CV than without (median difference of -20 seconds, or a median speed-up of 40.7%). This difference is statistically significant (Wilcoxon signed rank test: $Z = -3.3, p = 9.6 \cdot 10^{-4}$). See Figure 5.

Task times were not normally distributed.⁴ However, they conformed to a lognormal distribution. Due to statistical concerns with the inappropriate application of log normalisation [13] we opted for a nonparametric test.

Cognitive load: Participants reported a lower cognitive load when using CV than without (median difference of -2.25; the TLX is a 21-point scale). This difference is statistically significant (Wilcoxon signed rank test: $Z = -3.04, p = 0.0024$). Cognitive load scores were not normally distributed. See Figure 6.

⁴The Shapiro-Wilk test for normality was used throughout.

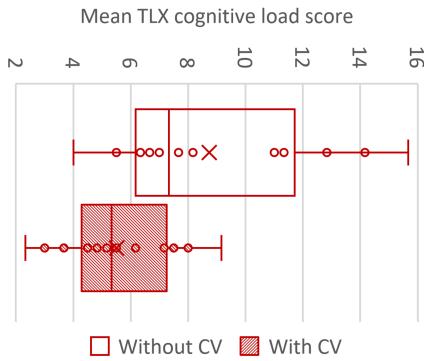


Fig. 6. Cognitive load scores with and without CV.

Analysing this result in terms of the six individual items on the TLX questionnaire, it appears as though this difference is attributable to three of them. With CV, there was a lower mental demand (median difference of -2.5), lower effort (median difference of -3), and lower frustration (median difference of -2.5). Of these, only the difference in frustration was statistically significant with Bonferroni correction applied (Wilcoxon signed rank test: $Z = -3.12, p = 0.0018$)

Self-efficacy: Participants had a slightly higher self-efficacy when using CV than without (median difference of 0.28; self-efficacy is a 10-point scale). This difference is not statistically significant. No individual item on the self-efficacy questionnaire showed significant differences between the with and without-CV conditions. We view this as a positive outcome, as it shows that the beneficial effects of shorter task times and lower cognitive load does not come at the cost of a reduced self-efficacy, which is sometimes the case when participants are asked to interact with a system that is more complex than what they are familiar with.

Effect of previous spreadsheet experience: participants were categorised into two groups based on their responses to the spreadsheet expertise self-assessment. Eleven participants fell into a ‘higher’ expertise group (**H**) and the other 11 into a ‘lower’ expertise group (**L**). Higher expertise was characterised by a prior knowledge of spreadsheet features relevant to our tasks (formulas, range notation, and drag-filling) as well as practical experience in applying these features. Lower expertise participants lacked knowledge, experience, or both.

While both H and L participants reported lower cognitive load overall, seven H participants reported a lower physical demand with CV, in comparison to only three L participants. Most L participants did not perceive drag-filling as physically demanding, despite the fact that experienced participants typically have developed coping mechanisms to deal with large drag-fill operations (e.g., checking the ranges beforehand, zooming the spreadsheet outwards, making selections using keyboard shortcuts) that reduce the physical effort of drag-filling. This is attributable to the fact that H participants apply drag-fills more regularly and so are more sensitive to the reduction in physical effort afforded by CV.

Revisiting task times, it appears as though H and L participants benefited to a very similar extent for debugging tasks (36.7% median speed-up for group H, 44.28% median speed-up for group L). However, L participants benefited to a greater extent during authoring tasks (55.3% median speed-up for group L, versus only 13.5% median speed-up for group H). Again, this can be attributed to the fact that H participants had developed better coping mechanisms that allowed them to be more efficient at drag-filling operations.

We did not observe a statistically significant difference in self-efficacy scores within either group H or L in isolation.

VI. MULTIPLE REPRESENTATIONS IN SPREADSHEETS

Calculation View’s fundamental idea is simple: provide a view of a spreadsheet that is optimised for understanding and manipulating its computational structure. This apparently straightforward idea has revealed a complex design space, the surface of which we have only scratched. In this section we describe alternatives that we have considered, or which might be scope for future work.

Variations of range assignment

What if you want to assign a single formula to a non-rectangular range, or even to disjoint ranges? Since the comma operator already denotes range union in Excel, we could allow it on the left hand side of an assignment, thus:

B1:B10, C1:C5, D7 = SQRT(A1)

Excel’s drag-fill also allows for constructing sequences of numbers or dates, such as 1,3,5,7... in a range of cells; manually type the first few entries, select them, and drag-fill. In CV, this will appear as a large number of literal assignments, concealing the user intent. We might instead imagine using ellipsis as a notation to indicate sequence assignment:

B1:B10 = 1,3,5,7...

Similarly, imagine that the cells **A1** and **B1** contain two distinct formulas. The user may select *both* and drag-fill downwards to copy. In CV, the user would have to make two edits, since they are two separate range assignments. We might instead provide a notation to capture this, for instance:

A2:B10 = copy A1:B1

Variations on the editing experience

The current CV editor inhabits a space between textual programming and the grid, in order to improve usability for non-expert end-users. However, for experts (e.g., with programming experience), we could use an *existing, generic IDE framework* (e.g., Visual Studio Code) as the editor for CV, where the expert programmer could rely on familiar affordances, including syntax highlighting, auto-complete, etc.

Textual notations have the capacity to solve a certain set of problems in spreadsheet interaction, but alternative representations might be better suited for solving different kinds of problems. Some sketches are shown in Figure 7. For instance, the editor could employ a blocks-style visual language, which

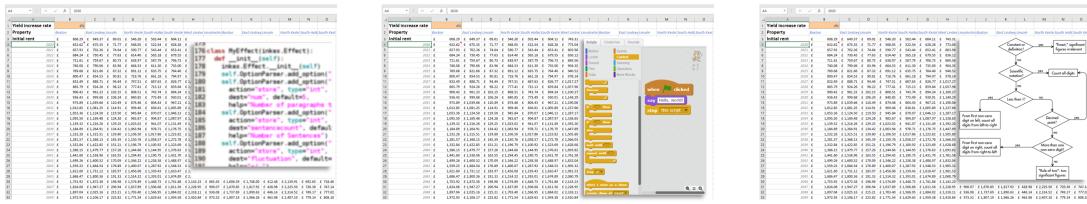


Fig. 7. Multiple representations need not just be text. From left to right: a professional code editor, a blocks programming language, and a flow chart.

would prevent syntactic errors. Alternatively, the editor could display formulas within a flow chart diagram, emphasising the dependencies between cells. In fact, the editor could display any number of spreadsheet-based visual programming languages, as long as the correspondence between the two representations was carefully considered. Users could then switch representations according to the task at hand.

Data specific to the alternative representation

Some content is present in the grid view, but not CV (e.g. cell formatting); but not the other way round. That is, the CV can be generated automatically, simply from the existing spreadsheet (Figure 3). However, a more expert programmer might want to do more in calculation view, such as using comments within formulas, and grouping together related assignments, even if they are not adjacent in the grid. In order to enable these types of secondary notation, additional information needs to be persisted within the file that is present in CV but not presented or editable in the grid.

Professionally written code is typically kept in a repository, and subject to code review, version control, and other engineering practices. If we could express *all* the information about a spreadsheet in textual form, these tools could also be applied to spreadsheets.

VII. RELATED WORK

A. Multiple representations and spreadsheet visualisation

Multiple representations have previously been applied in spreadsheets in the interactive machine learning domain [14], but not as simultaneous editing experiences. Programming languages theory has a concept of ‘lenses’ [15] which is a form of infrastructure enabling multiple representations. One application of lenses to spreadsheets [16] allows the user to edit the value of a formula, and have the edit propagate back to the cell’s input to the formula.

Previously explored approaches to mitigate computation hiding in spreadsheets include identification and visualisation of groups of related cells using colour [17]. Surfacing parts of the dataflow (cell dependency) graph, and allowing the graph to be directly manipulated, has also been explored [18]. Visualising the relationship between different sheets has also been shown to be beneficial [19]. Several commercial tools aim to assist with editing and debugging spreadsheet formulas, often via capabilities for visualisation.⁵

⁵Some examples include: www.arixcel.com, www.formuladesk.com, <https://devpost.com/software/formula-editor>, www.matrixlead.com

B. Overcoming spreadsheet errors

There are broadly two approaches to the mitigation of spreadsheet errors. The first approach is auditing tools, which rely on heuristics such as code smells [20], [21], [22] or type inference [23], [24], or assist users to write tests [25] to identify and report potential errors. They are not always effective [26], and they are limited by their post-hoc nature (i.e., they help users find errors after they have been made, rather than helping users avoid them in the first place), as well as their heuristics – they cannot detect errors not anticipated by developers of the tool. A machine learning approach where a model is trained on an error corpus [27] is unlikely to mitigate this latter limitation – here the heuristics are exemplified by the training dataset, rather than hand-coded.

The second approach to error mitigation in spreadsheets focuses on altering the structure of the grid, or creating an enhanced formula language. For example, sheet-defined functions [28] allow users to define custom functions in the grid. The Forms/3 system [29] focuses on the design space between grids and textual code. Representations such as hierarchical grids [30] support better object orientation in grids, sometimes combined with a richer formula language [31]. These formula languages can become sophisticated abstract specification languages that support ‘model-driven’ spreadsheet construction [32], [33], [34]. Excel’s ‘calculated columns’⁶ apply a single formula to an entire column, but using a more abstract ‘structured reference’ syntax, and there is no way to create a calculated ‘row’ or ‘block’. Excel’s array formulas⁷ use an abstract syntax to assign a single formula to a block of cells, but violate Kay’s ‘value principle’ [35] by forbidding inspection or editing of any of the constituent cells except the header. Solutions of this second approach address the poor *abstraction gradient* in spreadsheets [36], but require substantially greater expertise to use.

VIII. CONCLUSIONS AND NEXT STEPS

Our initial study has demonstrated that a textual calculation view of a spreadsheet, adjacent with the grid view, can make spreadsheets more comprehensible and maintainable. We plan to develop our prototype, exploring a number of variations, including using a free-form text editor, view filtering and navigational support, and enhanced syntax for assignments.

⁶<https://support.office.com/en-us/article/use-calculated-columns-in-an-excel-table-873fbac6-7110-4300-8f6f-aafa2ea11ce8>

⁷<https://support.office.com/en-us/article/create-an-array-formula-e43e12e0-afc6-4a12-bc7f-48361075954d>

REFERENCES

- [1] F. Hermans, B. Jansen, S. Roy, E. Aivaloglou, A. Swidan, and D. Hoepelman, "Spreadsheets are code: An overview of software engineering approaches applied to spreadsheets," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 5. IEEE, 2016, pp. 56–65.
- [2] R. R. Panko, "What we know about spreadsheet errors," *Journal of Organizational and End User Computing (JOEUC)*, vol. 10, no. 2, pp. 15–21, 1998.
- [3] T. Green and M. Petre, "Usability analysis of visual programming environments: a 'cognitive dimensions' framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131–174, 1996.
- [4] S. Ainsworth, "The functions of multiple representations," *Computers & education*, vol. 33, no. 2-3, pp. 131–152, 1999.
- [5] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman et al., "Scratch: programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.
- [6] A. Stead and A. F. Blackwell, "Learning syntax as notational expertise when using drawbridge," in *Proceedings of the Psychology of Programming Interest Group Annual Conference (PPIG 2014)*. Citeseer, 2014, pp. 41–52.
- [7] M. I. Gorinova, A. Sarkar, A. F. Blackwell, and D. Syme, "A live, multiple-representation probabilistic programming environment for novices," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 2016, pp. 2533–2537.
- [8] S. L. Tanimoto, "VIVA: A visual language for image processing," *Journal of Visual Languages & Computing*, vol. 1, no. 2, pp. 127–139, jun 1990.
- [9] A. Sfard, "On the dual nature of mathematical conceptions: Reflections on processes and objects as different sides of the same coin," *Educational studies in mathematics*, vol. 22, no. 1, pp. 1–36, 1991.
- [10] A. Sarkar, "The impact of syntax colouring on program comprehension," in *Proceedings of the 26th Annual Conference of the Psychology of Programming Interest Group (PPIG 2015)*, Jul. 2015, pp. 49–58.
- [11] S. G. Hart and L. E. Staveland, "Development of NASA-TLX (task load index): Results of empirical and theoretical research," in *Advances in psychology*. Elsevier, 1988, vol. 52, pp. 139–183.
- [12] D. R. Compeau and C. A. Higgins, "Computer self-efficacy: Development of a measure and initial test," *MIS quarterly*, pp. 189–211, 1995.
- [13] F. Changyong, W. Hongyue, L. Naiji, C. Tian, H. Hua, L. Ying et al., "Log-transformation and its implications for data analysis," *Shanghai archives of psychiatry*, vol. 26, no. 2, p. 105, 2014.
- [14] A. Sarkar, M. Jamnik, A. F. Blackwell, and M. Spott, "Interactive visual machine learning in spreadsheets," in *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. IEEE, Oct 2015, pp. 159–163.
- [15] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt, "Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 3, p. 17, 2007. [Online]. Available: <http://doi.acm.org/10.1145/1232420.1232424>
- [16] N. Macedo, H. Pacheco, N. R. Sousa, and A. Cunha, "Bidirectional spreadsheet formulas," in *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2014, Melbourne, VIC, Australia, July 28 - August 1, 2014*, S. D. Fleming, A. Fish, and C. Scaffidi, Eds. IEEE Computer Society, 2014, pp. 161–168. [Online]. Available: <https://doi.org/10.1109/VLHCC.2014.6883041>
- [17] R. Mittermeir and M. Clermont, "Finding high-level structures in spreadsheet programs," in *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*. IEEE, 2002, pp. 221–232.
- [18] T. Igarashi, J. D. Mackinlay, B.-W. Chang, and P. T. Zellweger, "Fluid visualization of spreadsheet structures," in *Visual Languages, 1998. Proceedings. 1998 IEEE Symposium on*. IEEE, 1998, pp. 118–125.
- [19] F. Hermans, M. Pinzger, and A. v. Deursen, "Detecting and visualizing inter-worksheet smells in spreadsheets," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 441–451.
- [20] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting and refactoring code smells in spreadsheet formulas," *Empirical Software Engineering*, vol. 20, no. 2, pp. 549–575, 2015.
- [21] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.
- [22] J. Zhang, S. Han, D. Hao, L. Zhang, and D. Zhang, "Automated refactoring of nested-if formulae in spreadsheets," *CoRR*, vol. abs/1712.09797, 2017. [Online]. Available: <http://arxiv.org/abs/1712.09797>
- [23] R. Abraham and M. Erwig, "Header and unit inference for spreadsheets through spatial analyses," in *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*. IEEE, 2004, pp. 165–172.
- [24] T. Cheng and X. Rival, "Static analysis of spreadsheet applications for type-unsafe operations detection," in *European Symposium on Programming Languages and Systems*. Springer, 2015, pp. 26–52.
- [25] A. Wilson, M. Burnett, L. Beckwith, O. Granatir, L. Casburn, C. Cook, M. Durham, and G. Rothermel, "Harnessing curiosity to increase correctness in end-user programming," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2003, pp. 305–312.
- [26] S. Aurigemma and R. Panko, "Evaluating the effectiveness of static analysis programs versus manual inspection in the detection of natural spreadsheet errors," *Journal of Organizational and End User Computing (JOEUC)*, vol. 26, no. 1, pp. 47–65, 2014.
- [27] R. Singh, B. Livshits, and B. Zorn, "Melford: Using neural networks to find spreadsheet errors," <https://www.microsoft.com/en-us/research/wp-content/uploads/2017/01/melford-tr-Jan2017-1.pdf>, 2017, last accessed 12 April 2018.
- [28] S. Peyton Jones, A. Blackwell, and M. Burnett, "A user-centred approach to functions in Excel," *ACM SIGPLAN Notices*, vol. 38, no. 9, pp. 165–176, 2003.
- [29] M. Burnett, J. Atwood, R. W. Djang, J. Reichwein, H. Gottfried, and S. Yang, "Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm," *Journal of functional programming*, vol. 11, no. 2, pp. 155–206, 2001.
- [30] K. S.-P. Chang and B. A. Myers, "Using and exploring hierarchical data in spreadsheets," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 2016, pp. 2497–2507.
- [31] D. Miller, G. Miller, and L. M. Parrondo, "Sumwise: A smarter spreadsheet," *EuSpRiG*, 2010.
- [32] J. Mendes, J. Cunha, F. Duarte, G. Engels, J. Saraiva, and S. Sauer, "Systematic spreadsheet construction processes," in *Visual Languages and Human-Centric Computing (VL/HCC), 2017 IEEE Symposium on*. IEEE, 2017, pp. 123–127.
- [33] M. Erwig, R. Abraham, S. Kollmansberger, and I. Cooperstein, "Gencel: a program generator for correct spreadsheets," *Journal of Functional Programming*, vol. 16, no. 3, pp. 293–325, 2006.
- [34] G. Engels and M. Erwig, "Classsheets: automatic generation of spreadsheet applications from object-oriented specifications," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 2005, pp. 124–133.
- [35] A. Kay, "Computer software," in *Scientific American*, vol. 251, no. 3, 1984, pp. 53–59.
- [36] D. G. Hendry and T. R. Green, "Creating, comprehending and explaining spreadsheets: a cognitive interpretation of what discretionary users think of the spreadsheet model," *International Journal of Human-Computer Studies*, vol. 40, no. 6, pp. 1033–1065, 1994.

No half-measures: A study of manual and tool-assisted end-user programming tasks in Excel

Rahul Pandita^{*†}, Chris Parnin[†], Felienne Hermans[‡], Emerson Murphy-Hill[†]

^{*}Phase Change Software, Golden, CO, USA

[†]North Carolina State University, Raleigh, NC, USA

[‡] Delft University of Technology, Delft, Netherlands

Email: rpandita@phasechange.ai, cjparnin@ncsu.edu, f.f.j.hermans@tudelft.nl, emerson@csc.ncsu.edu

Abstract—The popularity of end-user programming has lead to diverse end-user development environments. Despite accurate and efficient tools available in such environments, end-user programmers often manually complete tasks. What are the consequences of rejecting these tools? In this paper, we answer this question by studying end-user programmers completing four tasks with and without tools. In analyzing 111 solutions to each of these tasks, we observe that neither tool use nor tool rejection was consistently more accurate or efficient. In some cases, tool users took nearly twice as long to solve problems and over-relied on tools, causing errors in 95% of solutions. Compared to manual task completion, the primary benefit of tool use was narrowing the kinds of errors that users made. We also observed that partial tool use can be worse than no tool use at all.

I. INTRODUCTION

End-user programming [10] environments are designed to empower people without the significant knowledge of programming languages to efficiently perform tasks. These people can take advantage of automated procedures to solve problems that might otherwise have to be performed manually. Such tools come in many forms: as shortcuts and macros in editors, as stand-alone command-line programs, or formula operations in spreadsheets. Both research and practice suggest that certain tools can improve software quality and reduce development time (for example, Ko and Myers' Whyline [24]).

Despite the availability of tools and evidence that they can help, even professional developers oftentimes perform tasks manually instead of leveraging a tool. For instance, Murphy-Hill and colleagues report that programmers performed 90% of refactorings manually, despite having refactoring tools easily available [30]. Data scientists often adopt workflows that involve many manual steps when performing tasks such as data collection, data cleaning, and analysis [17]. However, manual task completion is not without consequences; for instance, developers make more errors when refactoring manually [15]. In contrast, improper tool configuration can also cause errors. For instance, unintended formatting of gene data in Excel has lead to widespread error in many scientific publications [36].

How do developers decide whether to use a tool or perform the task manually? Tversky and colleagues [34], [33] theorized that unlike automated systems that arrive at a decision based on an objective measure of probabilities derived by compounding individual simple probabilities, humans decisions are based on simple heuristics. These heuristics may interfere with a

developer's ability to accurately estimate the payoff in using a tool versus the risk of introducing manual errors. Similarly, Blackwell and Green's attention investment model [7], [6] explains the decision to invest in learning a new tool or skill based on several factors such as risk and expected payoff. These theories suggest that end-user programmers may make poor decisions about the risk of performing manual work and the perceived effort and benefit in learning to use a tool.

In this paper, we study *what contributes to programmers' decision to do manual work when the option to automate exists*. We enlisted online participants for four data extraction and data calculation tasks, and analyzed 111 responses for each of the tasks. Our main contribution is a study that explores the effectiveness and efficiency of end-user programming task performance with varying levels of automation.

From the study, we observed that neither complete automation nor a manual approach was consistently more accurate or efficient. We also observed that partial automation is sometimes worse than a manual approach. Additionally, many participants reported that although manual approach was not ideal they did not know what (and how) tools to leverage for automation. We found that most of the behavior could be explained by people either underestimating or overestimating factors related to risk and configuration effort in using tools. We recommend several design guidelines that improves the ability for users to estimate the effort involved in finding and learning tools for solving a given problem.

II. STUDY DESIGN

We analyzed tasks performed in Microsoft Excel because it is one of the most widely used programming environments, with 1.2 billion users [3]. Furthermore, Excel users have access to a wide spectrum of tools or automation options, from traditional programming in the form of Visual Basic for Applications (VBA) to Excel formulas to commands like filtering and sorting. For this study, we consider use of formulae, macros, and menu functions in Excel as tool use.

Our study seeks to answer the following research questions:

- 1) How did end-users solve the tasks?
- 2) How effective is manual effort versus automation, in terms of time and error rates?
- 3) What factors influence the choice of problem-solving strategy?

A. Participants

We recruited participants via Amazon Mechanical Turk (MTurk) [1], an on-line crowd-sourcing marketplace to facilitate and coordinate human intelligence tasks (HITs). Recent research shows that MTurk is an appropriate place to recruit study participants for behavioral research [28]. We recruited participants who have completed at least 1000 prior HITs as vetting criteria for reliability and experience of the participant. We paid participants \$3 USD for completing a set of tasks. We required participants to have Excel installed on their system.

B. Tasks

We designed tasks that: can be done in a variety of ways, including manually; cannot be completely solved with one tool; and reflect common tasks, namely *reading and extracting*, and *searching and filtering* [31]. We designed two tasks (1 and 2), each with two sub-tasks (A and B), and did not control for the order in which participants performed sub-tasks:

1) *Task 1A*: This sub-task required participants to extract zip codes from 61 lines of text, each line containing a postal address. Zip codes appeared strictly at the end of the text. A typical US zip code is 6 digits long. However, to break regularity, two zip codes contained 9 digits and one address contained a Canadian alphanumeric zip code. Figure 1a shows a list of addresses and extracted zip codes.

2) *Task 1B*: This sub-task required participants to extract zip-codes from 40 lines of text. The text in each line was a concatenation of name, postal address, phone number, and email. In this sub-task the zip code appeared interleaved in the text instead of at the end.

3) *Task 2A*: This sub-task presented participants with a list of words consisting of names of 229 fruits and vegetables (Figure 1c). We asked participants to count the number of words starting with “A”, starting with “B”, starting with “P” (deliberately not “C” to break regularity), and finally count the words containing “berries”.

4) *Task 2B*: This sub-task presented participants with a list of 1011 numbers, each representing the average number of hours a person sleeps. We asked participants to count the number of people that sleep 3–5 hours, that sleep 6–7 hours, and that sleep 8–9 hours.

C. Procedure

In a survey, participants were asked about their familiarity with Excel, from “Not at all familiar” to “Extremely familiar.” To help analyze how participants completed the tasks, we instructed participants to record macros, which recorded all Excel actions performed by participants. Participants could choose any strategy to solve the tasks. We also instructed participants to record the time spent on each sub-task, since macros do not capture timing. We provided participants with an Excel file containing a task to be performed. Although participants were free to finish the task at their own pace, they had to submit their solution within one hour to qualify for the compensation. Finally, we provided participants with a

B21		<input type="button" value="x"/>	<input type="button" value="v"/>	<input type="button" value="fx"/>	=RIGHT(A21,5)
	A				
19	129 S PROSPECT ST BOWLING GREEN OH 43403				43403
20	13 Gainsborough Ct., Ste. 147, Manalapan, NJ 07726				07726
21	1301 E WOOSTER ST BOWLING GREEN OH 43403				43403
22	1305 E WOOSTER ST BOWLING GREEN OH 43403				43403
23	1309 E WOOSTER ST BOWLING GREEN OH 43403				43403

(a) Task 1A: Extract zip codes from address strings.

```
Sub Task2()
    ActiveCell.FormulaR1C1 = "=COUNTIF('Task 1'!R[-2]C[-1]:R[226]C[-1], ""A*""")
    Range("B3").Select
    ActiveCell.FormulaR1C1 =
        "=COUNTIF('Task 1 - Data'!R[-2]C[-1]:R[226]C[-1], ""A*"""
    Range("B3").Select
    Selection.AutoFill Destination:=Range("B3:B6"), Type:=xlFillDefault
    Range("B3:B6").Select
    Range("B4").Select
    ActiveCell.FormulaR1C1 =
        "=COUNTIF('Task 1 - Data'!R[-3]C[-1]:R[225]C[-1], ""B*"""
    Range("B5").Select
    ActiveCell.FormulaR1C1 =
        "=COUNTIF('Task 1 - Data'!R[-4]C[-1]:R[224]C[-1], ""P*"""
    Range("A6").Select
    ActiveCell.FormulaR1C1 = "Count of Words Containing ""berries"""
    Range("A6").Select
    ActiveCell.FormulaR1C1 =
        "=COUNTIF('Task 1 - Data'!R[-5]C[-1]:R[223]C[-1], """*berries*"""
    Range("C11").Select
    ActiveWorkbook.Save
End Sub
```

(b) A recorded macro describing a participant’s task solution.

C3		
A	B	C
1 Apricots	A	
2 Asian Pear	A	12
3 Barbados Cherries	B	34
4 Black Currants	B	23
5 Blackberries	B	

(c) Task 2A: Counting prefixes and substrings in words.

Fig. 1: Screenshots of Tasks and Macro in Excel

link to upload their completed Excel files. They also answered the following questions about each sub-task:

- 1) About how long did the sub-task take you?
- 2) How did you approach the sub-task? Which tools, features, or functions did you use to help?
- 3) Do you think this is the most efficient strategy to solve the sub-task? If not, what prevented you from using a more efficient strategy?
- 4) How might you change your strategy if there were many more items to process?
- 5) Did you search online for help for this sub-task? What phrases did you search for?

D. Analysis

1) *Cleaning Data*: We first removed incomplete data. 254 people attempted Task 1 and 260 attempted Task 2. After excluding data where participants did not record macros, we selected 111 participants for each task. We did not necessar-

ily have 222 participants, since some participants may have independently completed both tasks.

2) *Identifying Strategies:* We analyzed the submitted Excel files and macros to extract the following information: the tools (functions or commands) used by the participant; the number of correctly answered questions; and any mistakes the participants made. In Figure 1b, we display an example task recording. We also recorded the self-reported time spent by participants for each sub-task. Finally, we classified each attempted sub-task into one of the following strategies:

- **Manual:** The participant does not use any tool.
- **Fully-automated:** The participant exclusively uses tools.
- **Semi-automated:** The participant uses a mix of the above strategies.

III. RESULTS

A. How did people solve the tasks?

Overall, we were surprised with the variety and creativity of solutions that participants used. Some participants wrote VBA scripts. Others used semi-automated techniques, such as first sorting numbers, and then manually selecting rows to get a count. A few participants used creative solutions, such as using the find-and-replace command for Task 2A, then using the resulting popup box that tells you the numbers of items that were replaced. Many participants used formulas to calculate solutions. For example, for Task 1A, it was popular to use a function like `RIGHT` to extract the last 5 digits to get the zipcode. For Task 2B, it was popular to use `COUNTIF` to count the number of items that met the search criteria.

A description of all the strategies that participants used and their frequency of use is on FigShare [2]. The number of solutions for each task ranged 5–8. For instance, one participant describes his approach for Task 1B as:

Again examined the data to look for a pattern I could use to extract the data. Used the Formula ribbon descriptions to get the proper search function (ie, find, search or lookup). Used MID and SEARCH to retrieve the 5 digit zip code. Visually inspected the results and found that my original pattern (search for " ???-") did not work for one record. Changed the pattern for the search to get the desired result.

Grouping these strategies by level of tool use, we can see in Table I how often participants decided to manually perform a task or use a tool instead. The use of manual or tool-assisted strategies greatly varied with the task performed. Many participants performed the zipcode extraction task manually (42.3% for Task 1A and 53.2% for Task 1B). In contrast, very few participants performed the counting tasks manually (4.5% for Task 2A and 3.6% for Task 2B). Instead, participants performed the counting tasks in a semi-automated manner (56.8% for Task 2A and 55.9% for Task 2B), using a tool to start but then finishing the calculation manually.

While analyzing macros we observed that participants spent a significant fraction of effort on browsing and getting familiar with the data. We computed the effort spent on

TABLE I: Manual and tool-assisted strategy usage rates

Task	Automated (%)	Semi (%)	Manual(%)
Task 1A	23 (20.7%)	41 (36.9%)	47 (42.3%)
Task 1B	25 (22.5%)	27 (24.3%)	59 (53.2%)
Task 2A	43 (38.7%)	63 (56.8%)	5 (04.5%)
Task 2B	45 (40.5%)	62 (55.9%)	4 (03.6%)

browsing by counting the fraction of macro statements that correspond to browsing (scrolling and selection) in Excel. For scrolling, we identified macro statements starting with : 1) “`ActiveWindow.Scroll`”, 2) “`ActiveWindow.SmallScroll`” or 3) “`ActiveWindow.LargeScroll`”. For selection, we identified macro statements ending with “`Select`” such as `Range("D230").Select`

On average 65% statements corresponded to browsing data. We also observe that the fraction of browsing statements depend on task, strategy, and correctness ($p < .001, \chi^2$). Participants browsed more in Task 2 than in Task 1; this finding can be explained by the larger amount of data that participants needed to process in Task 2. Surprisingly, participants who adopted a manual strategy for the tasks had *fewer* browsing statements than participants who used tools; we hypothesize that this may be because participants needed to inspect the data to understand its structure before applying a tool and also needed to inspect the data after the tool was applied to ensure correctness. Finally, participants who performed the task correctly tended to browse more; this could be explained by them taking extra care in reviewing their results.

Participants completed tasks in a variety of ways, including by repurposing tools. Furthermore, participants spent significant effort on browsing data, which correlated with task, strategy, and correctness.

B. Automated vs. manual performance

1) *How fast were people at solving the task?:* Mean times for all participants and strategies are presented in Figure 2. The length of each bar represents the time spent on a task using a strategy. One overall conclusion is that there is no consistently superior strategy. For example, on average to perform Task 1B, participants spent 412 seconds to do so manually, 564 seconds semi-automatically, and 790 seconds fully automatically manner. However, for Task 1A, we participants took 360 seconds to perform the task in automated manner, 416 seconds manually, and 564 seconds semi-automatically. Finally, although we report manual times for participants that did Task 2A ($n = 5$) and Task 2B ($n = 4$), these are primarily outliers who submitted completely incorrect answers.

Interestingly, the slowest performers in all tasks used tools exclusively to solve tasks. There were a handful of individuals who used fully automated solutions to solve the problem quickly, but the performance gain was only moderate over other manual users who were almost as fast. Why were tool users often so slow? According to self-reports in the post-survey, participants spent significant time trying to understand

TABLE II: Accuracy rates by strategy and task.

Task	Automated (%)	Semi (%)	Manual (%)
Task 1A	4.4%	31.7%	44.7%
Task 1B	88.0%	40.7%	44.0%
Task 2A	53.4%	34.9%	40.0%
Task 2B	71.1%	69.4%	25.0%

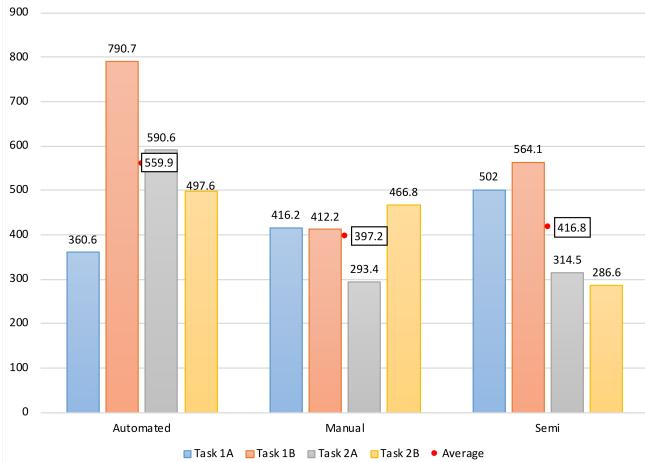


Fig. 2: Average time (in seconds) versus strategy. The • indicates the average time across all tasks.

or adapt the tool to the problem. Some slow performance in manual and semi-automated approaches could be explained by participants that attempted a more automated solution, but then abandoned their approach and completed the task in a manual fashion. One participant describes this situation:

Task 2A: Seemed straightforward, thought about trying a couple things, then figured I was overthinking it. I just did a sort and then highlighted the items and looked at the count.

There was no consistently fastest strategy, but tool-only users were often the slowest performers.

2) *How correct were people in solving the task?:* For this analysis, we measured correctness as the percentage of correct answers provided for a sub-task. We observed that, although tools are generally designed to reduce errors, participants who exclusively used tools were not immune to errors. In fact, many times participants were not only slower using tools, but wrong as well. For example, almost every participant using a tool (96%) made errors for Task 1A. Semi-automated approaches did not fare much better; they had the lowest accuracy ratings for three of the tasks.

Table II displays the accuracy of each strategies for each task. Task 1 has statistically significant differences in accuracy rates by strategy ($p < .01, \chi^2$), but not for Task 2. In Task 1A, automated solutions achieve a low accuracy rate of 4.4%, whereas in Task 1B, they have twice the accuracy rate over manual or semi-automated strategies.

Table III displays a breakdown of speed, accuracy, and

strategy. We used any task completion speed below the first percentile for a rating of *slow*, and any task completion speed above the third percentile for a rating of *fast*. If a participant made no errors, we indicate this as *correct*, otherwise, if they made any error, we indicate this as *error*. From this data, we can observe that it was not typical to have a fast solution and be correct. Unfortunately, participants that try to automate their solutions with tools are often still slow and incorrect. Further, no strategy consistently ensured both speed and correctness.

We also analyzed the correctness of task against the participant familiarity with Excel. Figure 3 plots the correctness against familiarity for each strategy. We next plot a regression line for each strategy using LOESS smoothing [11]. Based on LOESS analysis, using a fully-automated strategy results higher correctness across all familiarity levels. Another interesting trend is, while correctness in manual strategy increases as the familiarity increases, there is slight decline in overall correctness as familiarity increase for fully-automated and semi-automated strategies. This slight decline in overall correctness in fully and semi automated strategies may be attributed to blindspots introduced by tool use and familiarity.

Why did participants make so many errors with tool-assisted approaches, especially for Task 1A? One major reason was that participants may have failed to exercise any oversight. While most zip codes were 5 digits, a couple were 9 digits and one was alphanumeric. If participants assumed that all the zip-codes were 5 digit numbers and overlooked exceptions, they can very easily make this mistake. Given that many participants went on to do Task 1B, correctly with a tool, we suspect this is the case. For participants that did notice an error, this was often a reason to switch from a fully automated solution to a semi-automated solution. For example:

This one was fairly easy, just needed to lookup right truncation. Of course I overlooked the plus four zips. But there were so few, I just corrected by hand. Still only took 8 minutes.

Although tools could help improve correctness, they could also introduce blindspots that contribute to devastating error rates. No strategy was consistently accurate.

3) *What type of errors did people make?:* We wanted to understand the variety of errors that people may make and relate them to different strategy usage. We expected the error categories to be unique to the strategy taken by participants. For each participant, we classified the error they made into a pool of error categories by manually inspecting the recorded macro. From this inspection, we were able to infer the error that participants made in their solutions.

After examining the category of errors participants made, the most clear result was that participants who exclusively used tools made the fewest kinds of errors. That is, although automated users still were inaccurate, the error they made was typically isolated to one specific class of errors, whereas participants who made use of manual or semi-automated solutions had a much wider set of errors made. For example,

TABLE IV: Error Categories

Task	Str.	Error Category						
		Manual	Typing	Copy	Partial	No Task	Sort	System
1	Man	9	28	17	14	4	1	0
	Semi	11	15	12	10	7	2	0
	Auto	18	1	1	2	4	1	0
2	Man	5	0	1	0	0	0	0
	Semi	54	0	4	0	0	0	3
	Auto	23	0	7	0	2	0	0

TABLE V: Participants responses for ‘‘Do you think this is the most efficient strategy to solve the task?’’

Task	Response	Automated	Semi	Manual
Task 1	Yes	28	24	49
	Maybe	6	10	4
	No	14	33	42
Task 2	Yes	61	78	3
	Maybe	7	16	2
	No	20	23	4
Total		136	184	104

we systematically went through the post-survey responses. In particular, we analyzed participant responses for question: “*Do you think this is the most efficient strategy to solve the task? If not, what prevented you from using a more efficient strategy?*”

If the participant indicated that their approach was the most efficient for a subtask we classified the response as *Yes*, if not we classified the response as *No*. If the participants response indicated that (s)he was not sure we classified the response as *Maybe*. We did not consider 20 instances of empty or non-applicable responses across subs-tasks.

Table V presents the distribution of responses across Tasks for automated, semi, and manual approaches. Overall, 54% of participants responded *Yes*, 31% responded *No*, 10% responded *Maybe*, and 5% responded *NA* or did not respond. In all, 65% (89/136) of people that followed an automated strategy responded that their strategy was efficient. We were surprised that roughly half (52/104) the people that attempted the task manually also felt their strategy was efficient as well. The responses alluded to the fact that participants felt that the manual strategy was efficient because, it was simple and fast for the small dataset in the tasks, as captured in this response by a participant “*I think this is a pretty quick and dirty way of accomplishing the goal.*”

We explicitly asked participants to document what prevented them from using an efficient strategy, if they thought their strategy was not efficient. Two authors independently coded a random subset (10%) of responses. Authors followed the guidelines of open card sort [12], where they created categories based on the data itself. We then compared the results and documented that the authors were in agreement for 77.3% of their classifications. Based on the discussions, the first author then coded the rest of the responses. No new category emerged as the first author coded rest of the responses.

We next list the categories that emerged from the participant

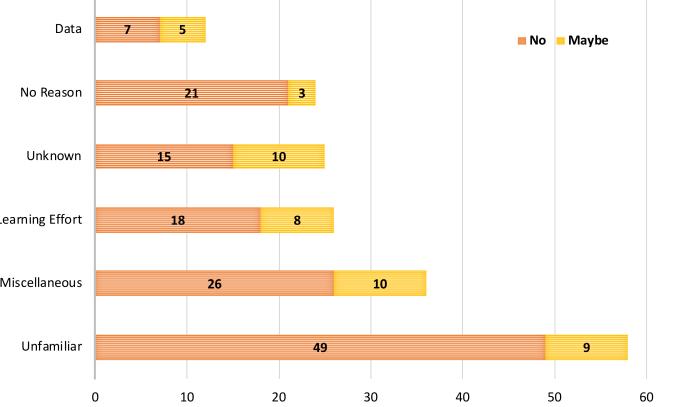


Fig. 4: Reasons for not selecting the most efficient strategy

responses to the question “*If not, what prevented you from using a more efficient strategy?*”:

- **Unknown:** Did not know a better way to do a task, e.g. “*no. I can't think of any other strategy*”
- **Unfamiliar:** The user is cognizant of the existence of tools but was not sure on what (or how) to use them, e.g. “*I am sure there was a formula that would have been faster, but I didn't know it.*”
- **Learning Effort:** The user was impeded by perceived effort in learning to use tool, e.g. “*could not figure out how to make it work*”.
- **Data:** The choice was dependent on data, e.g. “*For this set of data I think it was the fastest way to get it done*.”
- **Miscellaneous:** This was a catch-all category for responses that did not fit anywhere else. These included reasons such as: tool that did not work as expected, participant ran out of time, or participant made incorrect assumptions about the task.
- **No Reason:** Did not provide any reason or the provided reason is vague.

Figure 4 presents our findings on the impediments users face in employing what they consider as an efficient strategy. Most of the users were cognizant of existence of a tool that would allow them to perform the task better (Category “*Unfamiliar*”). However, they were not sure of either what tool to use or how to use a tool. Not considering the catch-all “*Miscellaneous*” category, the second most common impediment faced by participants was that they perceived the investment in learning about the tool too high for them to leverage tools in their task. Next participants reported that they were unaware of any other way of accomplishing the task. Followed by participants who thought their approach was not optimal, but data-set forced them to use the approach they chose.

While we anticipated participants to cite *unfamiliarity with tools* and *not knowing any other ways to solve a task* as the reasons for not using an optimal strategy, the explicit “*learning effort*” category provides opportunity for toolsmiths to design better tools that users perceive as easy to learn.

Users can be dissuaded from tool use based on the perceived learning effort.

IV. DISCUSSION

With the estimate that end-user developers outnumber professional developers by 50 million to 3 million [10], the goal of this research was to gain a deeper understanding of the decision process that end-user developers employ when deciding between manual effort or tool use. Additionally, some of our findings may generalize to professionals developers as well. This section discusses implications of our findings and threats to validity. We first summarise some of the general patterns we observed in the participant behavior.

A. Findings

1) No half-measures: We were surprised to find that participants performing the task either manually or in a complete automated fashion consistently outperformed participants employing a semi-automated approach, in dimensions of task correctness and speed. In general, a participant performing manual actions in the task is at a higher risk of introducing a mechanical errors [4]. However, we suspect that participants who performed the tasks manually were generally more careful to look for and avoid such errors. In contrast, participants performing the task in a semi-automated fashion may not have accounted for the risk of mechanical errors due to manual part of the strategy.

2) Tools reduce the kinds of errors made: We observed that the use of tools not only diminished the number of errors, they also helped participants avoid certain classes of errors. For instance, copy-paste and typing related errors are almost exclusively observed in the cases where participants attempted to perform the task manually or semi-automatedly. Such simple coding mistakes often produce notoriously difficult-to-find defects [20], [21].

Although use of tools did help participants avoid errors in general, Task 1A was an exception. Specifically, users of `RIGHT` function often did not correct for the interleaved exceptions (9 digit and Canadian zip codes). In this case, use of tool may have caused participants to get false sense of correctness by working 58 out of 61 cases and leaving out 3.

3) Large investments can go bust: From our analysis of strategy vs. time and correctness we observed that although some benefited from their investment to use tool, many often spent a long time learning how to get a tool to work and never received the expected payoff (either performed task very slow compared to others, or still made errors).

Further, not all solutions translated well from the first part of the task to the second part of the task. For example, 41 participants attempted Task 1A in using `RIGHT` function which does not lend itself to the Task 1B. In contrast, participants that used the `MID` function, were better able to adapt between tasks. When an investment went bust, participants would often just switch to a manual approach:

I couldn't find any way of easily doing the second task and as I had already spent so long on the first I just manually copied and pasted everything I could.

4) Tool selection factors: There is a large body of work in psychology that studies human decision making. For instance, Khaneman in his book “Thinking Fast and Slow” [23] talks about how human decision making is not always objective and is often affected by biases, beliefs, and heuristics. For instance *conjunction fallacy* [34] is phenomena when a person incorrectly assumes that specific conditions are more probable than a generic one. Another line of work that is relevant to this study is the *law of small numbers* [32] a form of *sampling bias*. When sampling, users focus on the little data at hand, while discounting issues which could occur in other data-sets. These effects were clear in our experiment, where participants were confident they were right even when there were better solutions. Concretely, we observed these effects in play when a significant number of participants incorrectly assumed that zip-codes are always 5 digit numbers on the right of the input string in Task 1A.

We also observed the *availability heuristic* [33] in participant behaviour. Availability heuristic causes a person to be more likely to weigh their judgment towards a recent event instead of objectively evaluating the present situation. We observed that most of the participants did not change their strategy for solving each subtask, even though they spent time adapting the strategies to the new subtask.

B. Implications for Design

The findings from the presented study may help with the design of the tools for facilitating better user interaction and engagement. We next outline some recommendations.

1) Provide estimates for learning effort: We observed that a significant number of participants had difficulty in realistically estimating the time and effort required on their part to understand and configure a tool, and whether that would be worth the investment.

Toolsmiths in a programming environment could assist their users to make better estimates. For instance, Viriyakattiporn and Murphy [35] proposed an approach to leverage a programmers history of tool use to actively recommend tool in current context. Likewise, Johnson and colleagues [22] propose leveraging developer knowledge to tailor a tools’ notifications.

An estimate of *difficulty* or *time-commitment* of using a tool can further enhance these approaches. For example, a naïve yet effective approach could be to provide an estimate of the time to configure a tool correctly based on how long other (first-time) users took to configure it. Furthermore, programming environments could attempt to actively guess what task users are attempting and provide them with contextual data.

2) Highlight unusual values after applying functions: We also observed that a significant number of participants in Task 1A incorrectly extracted the Canadian and the 9-digit zip codes. Partly because these participants approached the problem using the `RIGHT` function to extract the 5 characters

towards the right of the input string. While this approach worked well for 58 out of 61 cases, the participants still had to manually correct the one case involving the Canadian and two cases involving the 9-digit zip codes. Oftentimes, participants overlooked these exception cases and incorrectly reported the tool output as the correct zip-code.

While a manual inspection to verify tool output is highly recommended, we suggest tools should be preemptive in reminding developers to perform review. We also recommend toolsmiths to design tools cognizant of the “unusual” results to help ease the process of review. For instance, existing body of research on code-smells [13], [18], [19] can be extended to detect and report such instances to the user.

3) Tool Recommendation and Strategy Sharing: In the post-survey, when participants were asked to state the reasons that prevented them from using what they thought was an optimal strategy, 35 responses alluded to the fact participants were unaware of any other ways to solve the task at hand, despite a variety of strategies employed by other participants. Existing program synthesis approaches like FlashFill [16] in part alleviate the problem by automatically proposing a solution as a function of input output relationships. The programming environment can further help such users by recommending alternate strategies based on the current context of the user by leveraging the strategies employed by other users in similar contexts. For instance, environments can leverage concepts from “Programming by example” [26] where a software agent records the activities of users to reproduce them later. As the diversity of such recording grow overtime, these recordings can be queried as a shared resources (online or offline) for alternate strategy recommendation [29].

C. Threats to Validity

The primary threat to external validity is the representativeness of our data and tasks to the real world data cleaning workflows. To address this threat we focus on data extraction and data calculation tasks in Excel, which are typical computational tasks in programming and related fields such as data science. In 2014, New York Times reported [27] that analysts spend up to 80% of their time in cleaning data.

Threats to internal validity include the correctness of the identified functions used by the participants. Since authors manually identified the functions used by the participants, human error may affect our results. To minimize the effect, the authors checked the identified functions against the macros that were recorded by the participants while they were performing the tasks. Additionally, the authors manually coded the survey responses to identify the impediments faced by participants in using tools. To minimize this threat, we followed the safeguards for conducting empirical research proposed by Li [25]. To ensure researcher agreement about the findings, two authors independently analyzed the a random subset of participant responses to identify impediment categories.

These threats could be further minimized by evaluating more tasks in other developer programming environments and different settings. We plan to share various materials on the

project web [2], to enable other researchers to emulate our methods to repeat, refute, or improve our results.

V. RELATED WORK

Burnett and colleagues studied the introduction of a new tool for spreadsheets (assertions) that could be used to improve the detection of faults when compared to manual inspection [9]. Likewise, Cunha and colleagues demonstrate that Excel tools based on high level domain models help in avoiding errors [14]. In contrast, Murphy-Hill and colleagues demonstrate that users often do not use a specific set of tools to perform a specialized task (refactoring) [30]. However, we differ from this research in two different ways: 1) Instead of participants being instructed to use a specific tool, participants are given free reign to choose how they solve a task. This allows us to observe of this decision process. 2) Our tasks do not necessarily have a ready-made tool for directly solving the task (unlike performing an extract method refactoring with an extract method tool). Instead, participants must select from a federation of tools for solving the tasks, which sometimes involves manual steps in between application of two different tools.

To understand the decisions users make when selecting tools for problem-solving, Blackwell and Green [7] have proposed the investment of attention model. This framework describes four cost-benefit variables: cost, risk, investment, and payoff that help predict a person’s willingness to learn a new skill or try a new tool. Blackwell and Burnett [5] have used this to model adoption of a new tool in a spreadsheet. In a similar fashion, Brandt et al. introduce the concept of ‘Opportunistic Programming’[8], which describes a class of developers who adopt a minimum learning style and attempt to find online help specific for solving a task. Consistent with this approach, several participants in our study reported watching tutorial videos or reading blog posts in order to learn a strategy for solving the tasks.

VI. CONCLUSION

In this paper we found that participants performing Excel tasks using tools took more time on average than participants performing the task manually. However, 63% of participants performed the task correctly using automated solutions, compared to 37% participants that chose manual analysis. We found that most of the behavior could be explained by people either underestimating or overestimating factors related to risk and configuration effort in using tools. Environments that assist in estimating these factors may help future programmers make better choices when it comes to deciding whether to use tools.

ACKNOWLEDGMENT

This material is based upon work supported with funding from the Laboratory for Analytic Sciences and the Science of Security Lablet. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of any entity of the United States Government.

REFERENCES

- [1] Amazon Mechanical Turk. <https://www.mturk.com/mturk/welcome>.
- [2] Project Website. https://figshare.com/projects/Excel_Study/36959.
- [3] Microsoft by the numbers, November 2014. https://news.microsoft.com/bythenumbers/ms_numbers.pdf.
- [4] B. Bishop and K. McDaid. An empirical study of end-user behaviour in spreadsheet error detection & correction. *arXiv preprint arXiv:0802.3479*, 2008.
- [5] A. Blackwell and M. Burnett. Applying attention investment to end-user programming. In *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, pages 28–. IEEE Computer Society, 2002.
- [6] A. F. Blackwell. First steps in programming: A rationale for attention investment models. In *Proceedings of the IEEE Symposia on Human Centric Computing Languages and Environments*, pages 2–10. IEEE, 2002.
- [7] A. F. Blackwell and T. R. Green. Investment of attention as an analytic approach to cognitive dimensions. In *Collected Papers of the 11th Annual Workshop of the Psychology of Programming Interest Group (PPIG-11)*, pages 24–35, 1999.
- [8] J. Brandt, P. J. Guo, J. Lewenstein, and S. R. Klemmer. Opportunistic programming: How rapid ideation and prototyping occur in practice. In *Proceedings of the 4th International Workshop on End-user Software Engineering*, pages 1–5. ACM, 2008.
- [9] M. Burnett, C. Cook, O. Pendse, G. Rothermel, J. Summet, and C. Wallace. End-user software engineering with assertions in the spreadsheet paradigm. In *Proceedings of the 25th international conference on Software engineering*, pages 93–103, 2003.
- [10] M. M. Burnett and B. A. Myers. Future of end-user software engineering: beyond the silos. In *Proceedings of the on Future of Software Engineering*, pages 201–211. ACM, 2014.
- [11] W. S. Cleveland and S. J. Devlin. Locally weighted regression: an approach to regression analysis by local fitting. *Journal of the American statistical association*, 83(403):596–610, 1988.
- [12] J. Corbin and A. Strauss. *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage publications, 2014.
- [13] J. Cunha, J. P. Fernandes, P. Martins, J. Mendes, and J. Saraiva. Smellsheet detective: A tool for detecting bad smells in spreadsheets. In *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 243–244. IEEE, 2012.
- [14] J. Cunha, J. P. Fernandes, J. Mendes, and J. Saraiva. Embedding, evolution, and validation of model-driven spreadsheets. *IEEE Tran. on Software Engineering*, 41(3):241–263, 2015.
- [15] X. Ge and E. Murphy-Hill. Manual refactoring changes with automated refactoring validation. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1095–1105, New York, NY, USA, 2014. ACM.
- [16] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–330. ACM, 2011.
- [17] P. Guo. Data science workflow: Overview and challenges’ acm blog, 30 october 2013, 2013.
- [18] F. Hermans, M. Pinzger, and A. v. Deursen. Detecting and visualizing inter-worksheet smells in spreadsheets. In *Proceedings of the 34th International Conference on Software Engineering*, pages 441–451. IEEE Press, 2012.
- [19] F. Hermans, M. Pinzger, and A. van Deursen. Detecting code smells in spreadsheet formulas. In *Proceedings of the 2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 409–418. IEEE, 2012.
- [20] W. S. Humphrey. *A discipline for software engineering*. Addison-Wesley Longman Publishing Co., Inc., 1995.
- [21] W. S. Humphrey. The personal software process (PSP). 2000.
- [22] B. Johnson, R. Pandita, E. Murphy-Hill, and S. Heckman. Bespoke tools: adapted to the concepts developers know. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 878–881. ACM, 2015.
- [23] D. Kahneman. *Thinking, fast and slow*. Macmillan, 2011.
- [24] A. J. Ko and B. A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 151–158. ACM, 2004.
- [25] D. Li. Trustworthiness of think-aloud protocols in the study of translation processes. *International Journal of Applied Linguistics*, 14(3):301–313, 2004.
- [26] H. Lieberman. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.
- [27] S. Lohr. NY Times - For big-data scientists, ‘janitor work’ is key hurdle to insights, 2014.
- [28] W. Mason and S. Suri. Conducting behavioral research on amazons mechanical turk. *Behavior research methods*, 44(1):1–23, 2012.
- [29] E. Murphy-Hill. Continuous social screencasting to facilitate software tool discovery. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1317–1320. IEEE Press, 2012.
- [30] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012.
- [31] P. Pirolli and S. Card. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In *Proceedings of international conference on intelligence analysis*, volume 5, pages 2–4, 2005.
- [32] A. Tversky and D. Kahneman. Belief in the law of small numbers. *Psychological bulletin*, 76(2):105, 1971.
- [33] A. Tversky and D. Kahneman. Availability: A heuristic for judging frequency and probability. *Cognitive Psychology*, 5(2):207–232, 1973.
- [34] A. Tversky and D. Kahneman. Judgment under Uncertainty: Heuristics and Biases. In *Utility, Probability, and Human Decision Making*, pages 141–162. Springer Netherlands, 1975.
- [35] P. Viriyakattiporn and G. C. Murphy. Improving program navigation with an active help system. In *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research*, pages 27–41. IBM Corp., 2010.
- [36] M. Ziemann, Y. Eren, and A. El-Osta. Gene name errors are widespread in the scientific literature. *Genome Biology*, 17(1):177, 2016.

APPINITE: A Multi-Modal Interface for Specifying Data Descriptions in Programming by Demonstration Using Natural Language Instructions

Toby Jia-Jun Li¹, Igor Labutov², Xiaohan Nancy Li³, Xiaoyi Zhang⁵, Wenze Shi³, Wanling Ding⁴, Tom M. Mitchell², Brad A. Myers¹

¹HCI Institute, ²Machine Learning Dept., ³Computer Science Dept., ⁴Information Systems Dept.

Carnegie Mellon University, Pittsburgh, PA, USA

{tobyli, ilabutov, tom.mitchell, bam}@cs.cmu.edu

nancylxh14@gmail.com, {wenzes, wanlingd}@andrew.cmu.edu

⁵Computer Science & Engineering

University of Washington

Seattle, WA, USA

xiaoyiz@cs.washington.edu

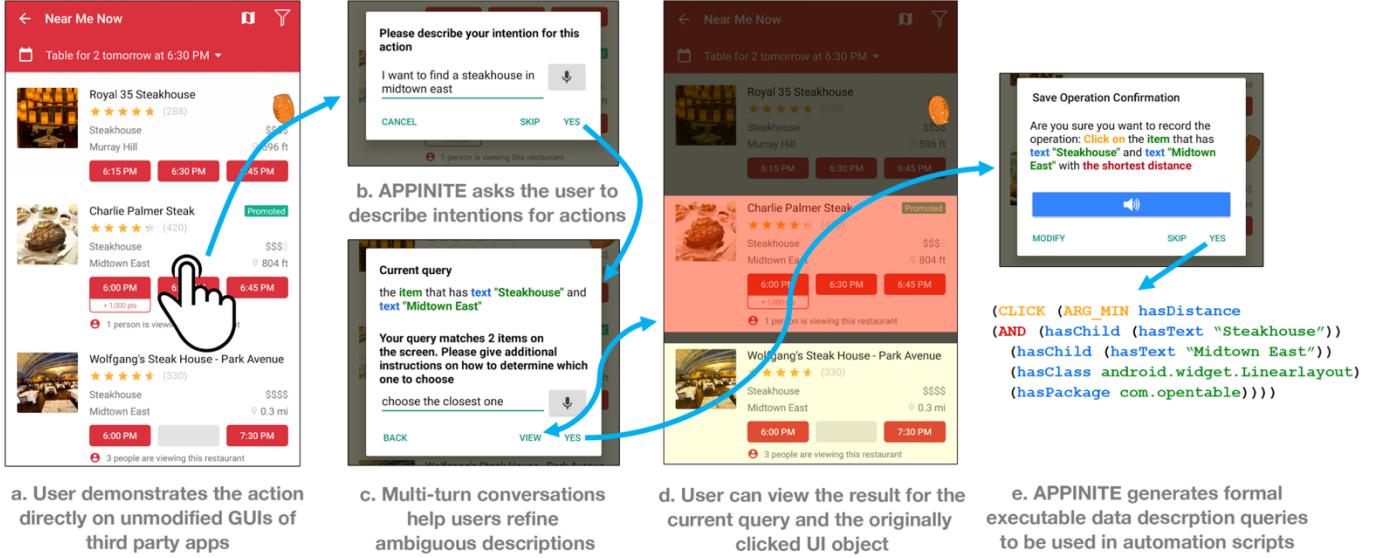


Fig. 1. Specifying data description in programming by demonstration using APPINITE: (a, b) enables users to naturally express their intentions for demonstrated actions verbally; (c) guides users to formulate data descriptions to uniquely identify target GUI objects; (d) shows users real-time updated results of current queries on an interaction overlay; and (e) formulates executable queries from natural language instructions.

Abstract—A key challenge for generalizing programming-by-demonstration (PBD) scripts is the *data description problem* – when a user demonstrates performing an action, the system needs to determine features for describing this action and the target object in a way that can reflect the user’s intention for the action. However, prior approaches for creating data descriptions in PBD systems have problems with usability, applicability, feasibility, transparency and/or user control. Our APPINITE system introduces a multi-modal interface with which users can specify data descriptions verbally using natural language instructions. APPINITE guides users to describe their intentions for the demonstrated actions through mixed-initiative conversations. APPINITE constructs data descriptions for these actions from the natural language instructions. Our evaluation showed that APPINITE is easy-to-use and effective in creating scripts for tasks that would otherwise be difficult to create with prior PBD systems, due to ambiguous data descriptions in demonstrations on GUIs.

Keywords—*programming by demonstration, end user development, verbal instruction, multi-modal interaction, natural language programming*

I. INTRODUCTION

Enabling end users to program new tasks for intelligent agents has become increasingly important due to the increasing ubiquity of such agents residing in “smart” devices such as phones, wearables, appliances and speakers. Although these agents have a set of built-in functionalities, and most provide expandability by allowing users to install third-party “skills”, they still fall short in helping users with the “long-tail” of tasks and suffer from the lack of customizability. Furthermore, many of users’ tasks involve coordinating the use of multiple apps, many of which do not even provide open APIs. Thus, it is unrealistic to expect every task to have a “skill” professionally made by service providers or third-party developers.

The lack of end-user programmability in intelligent agents results in an inferior user experience. When a user gives an out-of-domain command, the current conversational interface for most agents would either respond with a generic error message (e.g., “sorry, I don’t understand”) or perform a generic fallback action (e.g., a web search using the input as the search string). Often, neither response is helpful – a more natural and more useful response would be to ask the user to instruct the agent

This work was supported in part by Oath through the InMind project.

how to perform the new task [1]. Such end-user programmability also enables users to automate their repetitive tasks, reducing their redundant efforts.

Programming by demonstration (PBD) has been moderately successful at empowering end user development (EUD) of simple task automation scripts. Prior systems such as SUGILITE [2], PLOW [3] and CoScripter [4] allowed users to program task automation scripts for agents by directly demonstrating tasks using GUIs of third-party mobile apps or web pages. This approach enables users to program naturally by using the same environments in which they already know how to perform the actions, unlike in other textual (e.g., [5], [6]) or visual programming environments (e.g., [7]–[9]) where users need to map the procedures to a different representation of actions.

The central challenge for PBD is generalization. A PBD system should produce more than literal record-and-replay macros (e.g., sequences of clicks and keystrokes), but learn the task at a higher level of abstraction so it can perform similar tasks in new contexts [10], [11]. A key issue in generalization is the *data description problem* [10], [12]: when the user performs an action on an item in the GUI, what does it mean? The action and the item have many features. The system needs to choose a subset of features to describe the action and the item, so that it can correctly perform the right action on the right item in a different context. For example, in Fig. 1a, the user’s action is “Click”, and the target object can be described in many different ways, such as Charlie Palmer Steak / the second item from the list / the closest restaurant in Midtown East / the cheapest steakhouse, etc. The system would need to choose a description that reflects the user’s intention, so that the correct action can be performed if the script is run with different search results.

To identify the correct data description, prior PBD systems have varied widely in the division of labor, from making no inference and requiring the user to manually specify the features, to using sophisticated AI algorithms to automatically induce a generalized program [13]. Some prior systems such as SmallStar [12] and Topaz [14] used the “no inference” approach to give users full control in manually choosing features to use. However, this approach involves heavy user effort, and has a steep learning curve, especially for end users with little programming expertise. Others like SUGILITE [2], Peridot [15] and CoScipter [4] went a step further and used heuristic rules for generalization, which were still limited in applicability. This approach can only handle simple scenarios (unlike Fig. 1), and has the possibility of making incorrect assumptions.

At the other end of the spectrum, prior systems such as [16]–[20] used more sophisticated AI-based programming synthesis techniques to automatically infer the generalization, usually from multiple example demonstrations of a task. However, this approach has issues as well. It requires a large number of examples, but users are unlikely to be willing to provide more than a few examples, which limits the feasibility of this approach [21]. Even if end users provide a sufficient number of examples, prior studies [13], [22] have shown that untrained users are not good at providing *useful* examples that are meaningfully different from each other to help with inferring data descriptions.

Furthermore, users have little control of the resulting programs in these systems. The results are often represented in such a way that is difficult for users to understand. Thus, users cannot verify the correctness of the program, or make changes to the system [21], resulting in a lack of trust, transparency and user control.

In this paper, we present a new multi-modal interface named APPINITE¹, based on our prior PBD system SUGILITE [2], to enable end users to naturally express their intentions for data descriptions when programming task automation scripts by using a combination of demonstrations and natural language instructions on the GUIs of arbitrary third-party mobile apps. APPINITE helps users address the data description problem by guiding them to verbally reveal their intentions for demonstrated actions through multi-turn conversations. APPINITE constructs data descriptions of the demonstrated action from natural language explanations. This interface is enabled by our novel method of constructing a semantic relational knowledge graph (i.e., an ontology) from a hierarchical GUI structure (e.g., a DOM tree). We use an interaction proxy overlay in APPINITE to highlight ambiguous references on the screen, and to support meta actions for programming with interactive UI widgets in third-party apps.

APPINITE provides users with greater expressive power to create flexible programming logic using the data descriptions, while retaining a low learning barrier and high understandability for users. Our evaluation showed that APPINITE is easy-to-use and effective in tasks with ambiguous actions that are otherwise difficult or impossible to express in prior PBD systems.

II. BACKGROUND AND RELATED WORK

A. Multi-Modal Interfaces

Multi-modal interfaces process two or more user input modes in a coordinated manner to provide users with greater expressive power, naturalness, flexibility and portability [23]. APPINITE combines speech and touch to enable a “speak and point” interaction style, which has been studied since the early multi-modal systems like Put-that-there [24]. In programming, similar interaction styles have also been used for controlling robots (e.g., [25], [26]). A key pattern in APPINITE’s multi-modal interaction model is *mutual disambiguation* [27]. When the user demonstrates an action on the GUI with a simultaneous verbal instruction, our system can reliably detect *what* the user did and *on which* UI object the user performed the action. The demonstration alone, however, does not explain *why* the user performed the action, and any inferences on the user’s intent would be fundamentally unreliable. Similarly, from verbal instructions alone, the system may learn about the user’s intent, but grounding it onto a specific action may be difficult due to the inherent ambiguity in natural language. Our system utilizes these complementary inputs to infer robust and generalizable scripts that can accurately represent user intentions in PBD.

A unique challenge for APPINITE is to support multi-modal PBD on arbitrary third-party GUIs. Some of such GUIs can be highly complicated with hundreds of objects, each with many different properties, semantic meanings and relationships with other objects. Moreover, third-party apps only expose low-level hierarchical representations of their GUIs at the presentation

¹ APPINITE is a type of rock, and stands for Automation Programming on Phone Interfaces using Natural-language Instructions with Task Examples.

layer, without information about internal program logic or semantics. Prior systems such as CommandSpace [28], Speechify [29] and PixelTone [30] investigated multi-modal interfaces that can map coordinated natural language instructions and GUI gestures to system commands and actions. But the use of these systems are limited to specific first-party apps and task domains, in contrast to APPINITE which aims to be general-purpose.

B. Generalization and Data Description Problems in PBD

Having accurate data descriptions to correctly reflect user intentions in different contexts is crucial for ensuring the generalizability in PBD. Prior PBD systems range from making no inference at all to using sophisticated AI algorithms to infer data descriptions for demonstrated actions [13].

The “no inference” approach (e.g., [12], [14]) shows dialogs to ask users to make selections on feature(s) to use for data descriptions when ambiguities arise, which gives users full control but suffers in usability because end users may have trouble understanding and choosing from the options, especially when the tasks are complicated, or when their intentions are non-trivial. The AI-based program synthesis approach (e.g., [16]–[20]) requires a large number of examples to cover the space of different contexts to synthesize from, which is not feasible in many cases when end users are unwilling to provide sufficient number of examples [21], or unable to provide high-quality examples with good coverage [13], [22]. Users also have limited control and understanding of the inference and synthesis process, as AI-based algorithms used in these systems often suffer in explainability and transparency [21].

APPINITE addresses these issues by providing a multi-modal interface to specify data descriptions verbally through a multi-initiative conversation. It provides users with control and transparency of the process, retains usability by allowing users to describe the data descriptions in natural language, provides increased expressive power in parsing natural language instructions, and eliminates redundancy by only requiring one example of demonstration and instruction.

C. Learning Tasks from Natural Language Instructions

Natural language instruction is a common medium for humans to teach each other new tasks. For an agent to learn from such instructions, a major challenge is grounding – the agent needs to extract semantic meanings from instructions, and associate them with actions, perceptions and logic [31]. This process is also related to the concept of natural language programming [32]. Some prior work has tried translating natural language directly to code (e.g., [33]–[35]), but these systems required users to instruct using inflexible structures and keywords that resemble those of the programming languages, which made such systems unsuccessful for end user developers.

In specific task domains such as navigation [36], email [31], robot control [37] or basic phone operations [38], the number of relevant actions and concepts are small, which makes it feasible to parse natural language into formal semantic representations in a smaller space of pre-defined actions and concepts.

An effective way to constrain user natural language instructions, but still support a wide variety of tasks, is to leverage GUIs of existing apps or webpages. PLOW [3] is a web automation agent that uses GUIs to ground natural language instruction. It asks users to provide “play-by-play” natural

language instructions with task demonstrations, which is similar to APPINITE. PLOW grounds the instructions by resolving noun phrases to items on the screen through a heuristic search on the DOM tree of the webpage. SUGILITE [2], on the other hand, uses a single utterance describing the task from the user for each script to perform parameterization by grounding phrases in the initial utterance (e.g., order a cup of *cappuccino*) to a demonstrated action (e.g., select *cappuccino* from a list menu).

Compared with prior systems, APPINITE specifically focuses on helping users specify accurate data descriptions that reflect their intentions using a combination of natural language instructions and demonstrations. Our novel semantic relational graph representation of the GUI allows users to use a wider range of semantic (e.g. “*cheapest restaurant*”) and relational (e.g., “*score for Pittsburgh Steelers*”) expressions without being tied to the underlying GUI implementation. Users can also use more flexible logic in their instructions thanks to our versatile semantic parser. To ensure usability while giving the user full control, our mixed-initiative system can engage in multi-turn conversations with users to help them clarify and extend data descriptions when ambiguities arise.

III. FORMATIVE STUDY

We conducted a formative study to understand how end users may verbally instruct the system simultaneously while demonstrating using the GUIs of mobile apps, and whether these instructions would be useful for addressing the data description problem. We asked workers from Amazon Mechanical Turk (mostly non-programmers [39]) to perform a sample set of tasks using a simulated phone interface in the browser, and to describe the intentions for their actions in natural language. We recruited 45 participants, and had them each perform 4 different tasks. We randomly divided the participants into two groups. One group of participants were simply told to narrate their demonstrations in a way that would be helpful even if the exact data in the app changed in the future. Another group were additionally given detailed instructions and examples of how to write good explanations to facilitate generalization from demonstrations.

After removing responses that were completely irrelevant, or apparently due to laziness (32% of the total), the majority (88%) of descriptions from the group that were not given detailed instructions and all of descriptions (100%) from the group that received detailed instructions explained intentions for the demonstrations in ways that would facilitate generalization, e.g., by saying “*Scroll through to find and select the highest rated action film, which is Dunkirk*” rather than just “*select Dunkirk*” without explaining the characteristic feature behind their choice.

We also found that many of such instructions contain spatial relations that are either explicit (e.g., “*then you click the back button on the bottom left*”) or implicit (e.g., “*the reserve button for the hotel*”, which can translate to “*the button with the text label ‘reserve’ that is next to the item representing the hotel*”). Furthermore, approximately 18% of all 1631 natural language statements we collected from this formative study used some generalizations (e.g., the highest rated film) in the data description instead of using constant values of string labels for referring to the target GUI objects. These findings illustrate the need for constructing an intermediate level representation of GUIs that

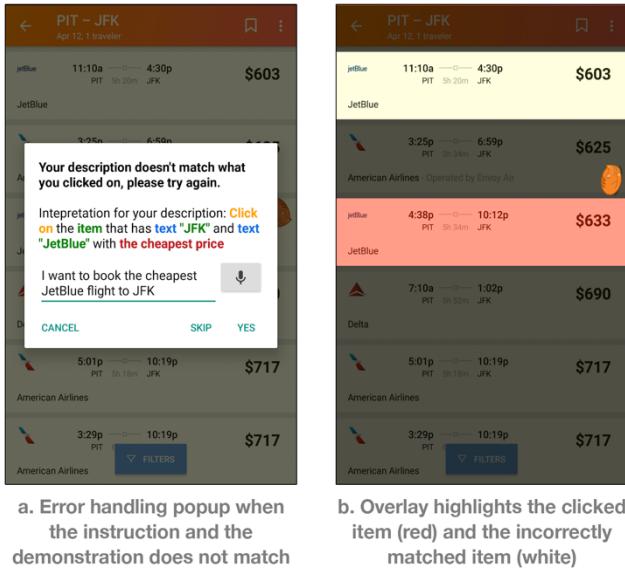


Fig. 2. APPINITE’s error handling interfaces for handling situations where the instruction and the demonstration do not match.

abstracts the semantics and relationships from the platform-specific implementation of GUIs and maps more naturally to the semantics of likely natural language explanations.

IV. THE APPINITE INTERFACE

Informed by the results from the formative study, we have designed and implemented APPINITE to enable users to provide natural language instructions for specifying data descriptions in PBD. It uses our open-source SUGILITE [2] framework for detecting and replaying user demonstrations. APPINITE aims to improve the process for specifying data descriptions in PBD through its novel multi-modal interface, which provides end users with greater expressive power to create flexible programming logic while retaining a low learning barrier and high understandability. In this section, we discuss the user experience of APPINITE with an example walkthrough of specifying the data description for programming a script for making a restaurant reservation using the OpenTable app. Readers can also refer to the supplemental video figure for a similar example task.

A. APPINITE User Experience

After the user starts a new demonstration recording, she demonstrates clicking on the OpenTable icon on the home screen, and chooses the “Near Me Now” option on the main screen of OpenTable, which are exactly the same steps that she would do normally to make a restaurant reservation. Neither of these steps is ambiguous, because their data descriptions (clicking on the icon / FrameLayout object with text labels “OpenTable” / “Near Me Now”) can be inferred using heuristic rules. Thus, the APPINITE disambiguation feature will not be invoked. Instead, the user directly confirms the recording either by speech or by tapping on a popup (Fig. 1e).

As the next action, the user chooses a restaurant from the result list (Fig. 1a). This action is ambiguous because its target UI object has multiple reasonable properties for data description, for which the heuristic-based approach cannot determine which

one would reflect the user’s intention. Therefore, APPINITE’s interaction proxy overlay (details in Section V) prevents this tap from invoking the OpenTable app action, and asks the user, both vocally and visually through a popup dialog, to describe her intention for the action. The user can then either speak or type in natural language. Leveraging the UI snapshot graph extraction and natural language instruction parsing architecture (details in Section V), APPINITE can understand flexible data descriptions expressed in diverse natural language instructions. These descriptions would otherwise be impractical for end users to manually program. Below we list some example instructions that APPINITE can support for the GUI shown in Fig. 1a.

- *I want to choose the second search result*
- *Find the steakhouse with the earliest time available*
- *Here I’m selecting the closest promoted restaurant*
- *I will book a steakhouse in Midtown East*

End users might not be able to provide complete data descriptions to uniquely identify target UI objects on their first attempt. To address this issue, APPINITE uses a mixed-initiative multi-turn dialog interface to initiate follow-up conversations to help users refine data descriptions. For instance, as shown in Fig. 1c, the description parsed from the user’s instruction matches two items in the list. APPINITE asks the user what additional criteria can be used to choose between the GUI objects when the initial query matches multiple ones. The user can preview the result of executing the current query on a screen captured from the underlying app’s GUI (Fig. 1d). In this preview interface, the actually clicked object is marked in red, while the other matched ones (false positives) are highlighted in white. The user can iteratively refine the data description, add new requirements and preview the real-time result of the current data description until she has one that can both uniquely identify the action she has demonstrated and accurately reflects her intention.

Lastly, APPINITE formulates an executable data description query for the demonstrated action and adds it to the current automation script (Fig. 1e). This data description is used by the intelligent agent to choose the correct action to perform in future executions of the script in different contexts. The interaction proxy overlay then sends the previously held tap to the underlying app GUI, so that the app can proceed to the next step so the user can continue demonstrating the task.

In the above example, the user has interacted with the APPINITE interface in the “demonstration-first” mode where she first demonstrates the action, and only needs to provide natural language instructions to clarify her intention for the action if disambiguation is required. Alternatively, APPINITE’s multi-modal interface also supports a “verbal-first” mode where she can first describe the action in natural language, after which she would only be asked to tap the correct UI object for grounding the data description if her description is ambiguous and matches multiple objects. All APPINITE interfaces used for recording are also speech-enabled, where users can freely choose the most natural interaction modality for the context – either direct manipulation, natural language instruction or a mix of both.

APPINITE also provides end-user-friendly error messages when the user’s instruction does not match the demonstration

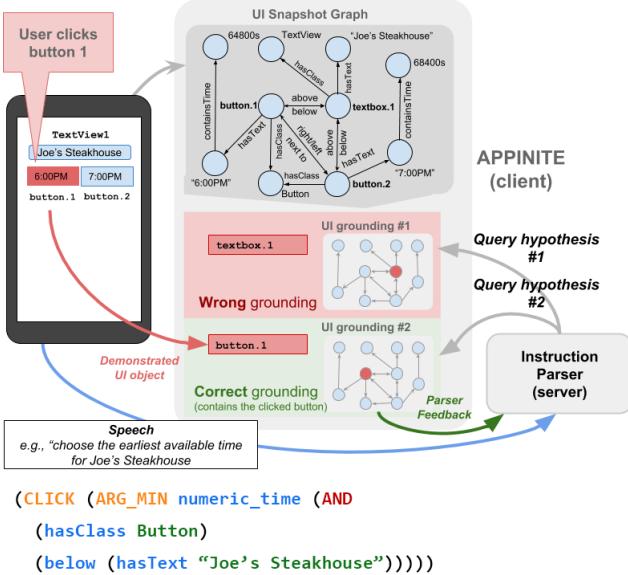


Fig. 3. APPINITE’s instruction parsing process illustrated on an example UI snapshot graph constructed from a simplified GUI snippet.

(Fig. 2), or when the parser fails to parse the user’s natural language instructions into valid data description queries. If a user has encountered the same error more than once, APPINITE switches to more detailed spoken prompts that ask the user to refer to contents and properties shown on the screen about the target UI object of the demonstrated action when describing the intention in natural language. Our user study showed that this helped users give successful descriptions (see Section VI).

At the end of the demonstration, the resulting script will be stored, and can later be invoked either using a GUI, from an external web service, by an IoT device or through an intelligent agent using the script execution mechanisms provided by the SUGILITE framework [2], [40]. The script can also be generalized (e.g., using a script demonstrated for making a reservation at a steakhouse to also make a reservation at a sushi restaurant) using script generalization mechanisms provided in SUGILITE [2].

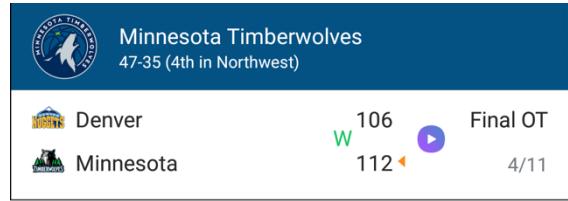
V. DESIGN AND IMPLEMENTATION

In this section, we discuss the design and implementation of three core components of APPINITE: the UI snapshot graph extractor, the natural language instruction parser, and the interaction proxy overlay.

A. UI Snapshot Knowledge Graph Extraction

We found in the formative study that end users often refer to spatial and semantic-based generalizations when describing their intentions for demonstrated actions on GUIs. Our goal is to translate these natural language instructions into formal executable queries of data descriptions that can be used to perform these actions when the script is later executed. Such queries should be able to generalize across different contexts and small variations in the GUI to still correctly reflect the user’s intentions. To achieve this goal, a prerequisite is a representation of the GUI objects with their properties and relationships, so that queries can be formulated based on this representation.

APPINITE extracts GUI elements using the Android Accessibility Service, which provides the content of each window in the current GUI through a static hierarchical tree representation [41]



```
(DEFINE score (AND (isNumeric true)
  (hasClass android.widget.TextView)
  (rightTo (AND (hasText "Minnesota")
    (hasClass android.widget.TextView))))))
```

Fig. 4. A snippet of an example GUI where the alignment suggests a semantic relationship – “*This is the score for Minnesota*” translates into ““Score” is the TextView object with a numeric string that is to the right of another TextView object ‘Minnesota.’”

similar to the DOM tree used in HTML. Each node in the tree is a *view*, representing a UI object that is visible (e.g., buttons, text views, images) or invisible (often created for layout purposes). Each view also contains properties such as its Java class name, app package name, coordinates for its on-screen bounding box, accessibility label (if any), and raw text string (if any). Unlike a DOM, our extracted hierarchical tree does not contain specifications for the GUI layout other than absolute coordinates at the time of extraction. It does not contain any programming logic or meta-data for the text values in views, but only raw strings from the presentation layer. This hierarchical model is not adequate for our data description, as it is organized by parent-child structures tied to the implementation details of the GUI, which are invisible to end users of the PBD system. The hierarchical model also does not capture geometric (e.g., next to, above), shared property value (e.g., two views with the same text), or semantic (e.g., the *cheapest* option) relations among views, which are often used in users’ data descriptions.

To represent and to execute queries used in data descriptions, APPINITE constructs relational knowledge graphs (i.e., ontologies) from hierarchical GUI structures as the medium-level representations for GUIs. These *UI snapshot graphs* abstract the semantics (values and relations) of GUIs from their platform-specific implementations, while being sufficiently aligned with the semantics of users’ natural language instructions. Fig. 3 illustrates a simplified example of a UI snapshot graph.

Formally, we define a UI snapshot graph as a collection of *subject-predicate-object* triples denoted as (s, p, o) , where the subject s and the object o are two entities, and the predicate p is a directed edge representing a relation between the subject and the object. In our graph, an entity can either represent a view in the GUI, or a typed (e.g., string, integer, Boolean) constant value. This denotation is highly flexible – it can support a wide range of nested, aggregated, or composite queries. Furthermore, a similar representation is used in general-purpose knowledge bases such as DBpedia [42], Freebase [43], Wikidata [44] and WikiBrain [45], which can enable us to easily plug our UI snapshot graph into these knowledge bases to support better semantic understanding of app GUIs in the future.

The first step in constructing a UI snapshot graph from the hierarchical tree extracted from the Android Accessibility Service is to flatten all views in the tree into a collection of view

entities. The hierarchical relations are still preserved in the graph, but converted into `hasChild` and `hasParent` relationships between the corresponding view entities. Properties (e.g., coordinates, text labels, class names) are also converted into relations, where the values of the properties are represented as entities. Two or more constants with the same value (e.g., two views with the same class name) are consolidated as a single constant entity connected to multiple view entities, allowing easy querying for views with shared properties values.

In GUI designs, horizontal or vertical alignments between objects often suggest a semantic relationship [5]. Generally, smaller geometric distance between two objects also correlates with higher semantic relatedness between them [46]. Therefore, it is important to support spatial relations in data descriptions. APPINITE adds spatial relationships between view entities to UI snapshot graphs based on the absolute coordinates of their bounding boxes, including `above`, `below`, `rightTo`, `leftTo`, `nextTo`, and `near` relations. These relations capture not only explicit spatial references in natural language (e.g., the button *next to* something), but also implicit ones (see Fig. 4 for an example). In APPINITE, thresholds in the heuristics for determining these spatial relations are relative to the dimension of the screen, which supports generalization across phones with different resolutions and screen sizes.

APPINITE also recognizes some semantic information from the raw strings found in the GUI to support grounding the user’s high-level linguistic inputs (e.g., “*item with the lowest price*”). To achieve this, APPINITE applies a pipeline of data extractors on each string entity in the graph to extract structured data (e.g., phone number, email address) and numerical measurements (e.g., price, distance, time, duration), and saves them as new entities in the graph. These new entities are connected to the original string entities by “`contains`” relations (e.g., `containsPrice`). Values in each category of measurements are normalized to the same units so they can be directly compared, allowing flexible computation, filtering and aggregation.

B. Instruction Parsing

After APPINITE constructs a UI snapshot graph, the next step is to parse the user’s natural language description into a formal executable query to describe this action and its target UI object. In APPINITE, we represent queries in a simple but flexible LISP-like query language (*S-expressions*) that can represent joins, conjunctions, superlatives and their compositions. Fig. 1e, Fig. 3 and Fig. 4 show some example queries.

Representing UI elements as a knowledge graph offers a convenient data abstraction model for formulating a query using language that is closely aligned with the semantics of users’ instructions during a demonstration. For example, the utterance “*next to the button*” expresses a natural *join* over a binary relation `near` and a unary relation `isButton` (a unary relation is a mapping from all UI object entities to truth values, and thus represents a subset of UI object entities.) An utterance “*a textbox next to the button*” expresses a natural *conjunction* of two unary relations, i.e., an intersection of a set of UI object entities. An utterance such as “*the cheapest flight*” is naturally expressed as a *superlative* (a function that operates on a set of UI object entities and returns a single entity, e.g., `ARG_MIN` or `ARG_MAX`). Formally, we define a data description query in our

language as an *S-expression* that is composed of expressions that can be of three types: *joins*, *conjunctions* and *superlatives*, constructed by the following 7 grammar rules:

$$\begin{aligned} E &\rightarrow e; \quad E \rightarrow S; \quad S \rightarrow (join \ r \ E); \quad S \rightarrow (and \ S \ S) \\ T &\rightarrow (ARG_MAX \ r \ S); \quad T \rightarrow (ARG_MIN \ r \ S); \quad Q \rightarrow S \mid T \end{aligned}$$

where Q is the root non-terminal of the query expression, e is a terminal that represents a UI object entity, r is a terminal that represents a relation, and the rest of the non-terminals are used for intermediate derivations. Our language forms a subset of a more general formalism known as Lambda Dependency-based Compositional Semantics [47] a notationally simpler alternative to lambda calculus which is particularly well-suited for expressing queries over knowledge-graphs.

Our parser uses a Floating Parser architecture [48] and does not require hand-engineering of lexicalized rules, e.g., as is common with synchronous CFG or CCG based semantic parsers. This allows users to express lexically and syntactically diverse, but semantically equivalent statements such as “*I am going to choose the item that says coffee with the lowest price*” and “*click on the cheapest coffee*” without requiring the developer to hand-engineer or tune the grammar for different apps. Instead, the parser learns to associate lexical and syntactic patterns (e.g., associating the word “*cheapest*” with predicates `ARG_MIN` and `containsPrice`) with semantics during training via rich features that encode co-occurrence of unigrams, bigrams and skipgrams with predicates and argument structures that appear in the logical form. We trained the parser used in the preliminary usability study via a small number of example utterances paired with annotated logical forms and knowledge-graphs (840 examples), using 4 of the 8 apps used in the user studies as a basis for training examples. We use the core Floating Parser implementation within the SEMPRE framework [49].

C. Interaction Proxy Overlay

Prior mobile app GUI-based PBD systems such as SUGILITE [2] instrument GUIs by passively listening for the user’s actions through the Android accessibility service, and popping up a disambiguation dialog *after* an action if clarification of the data description is needed. This approach allows PBD on unmodified third-party apps without access to their internal data, which is constrained by working with Android apps (unlike web pages, where run-time interface modification is possible [5], [50], [51]). However, at the time when the dialog shows up, the context of the underlying app may have already changed as a result of the action, making it difficult for users to refer back to the previous context to specify the data description for the action. For example, after the user taps on a restaurant, the screen changes to the next step, and the choice of restaurant is no longer visible.

To address these issues, we implemented an interaction proxy [52] to add an interactive overlay on top of third-party GUIs. Our mechanism can run on any phone running Android 6.0 or above, without requiring root access. The full-screen overlay can intercept all touch events (including gestures) before deciding whether, or when to send them to the underlying app, allowing APPINITE to engage in the disambiguation process while preventing the demonstrated action from switching the app away from the current context. Users can refine data descriptions

through multi-turn conversations, try out different natural language instructions, and review the state of the underlying app when demonstrating an action without invoking the action.

The overlay is also used for conveying the state of APPINITE in the mixed-initiative disambiguation to improve transparency. An interactive visualization highlights the target UI object in the demonstration, and matched UI objects in the natural language instruction when the user's instruction matches multiple UI objects (Fig. 1d), or the wrong object (Fig. 2a). This helps users to focus on the differences between the highlighted objects of confusion, assisting them to come up with additional differentiating criteria in follow-up instructions to further refine data descriptions. In the “verbal-first” mode where no demonstration grounding is available, APPINITE also uses similar overlay highlighting to allow users to preview the matched object results for the current data description query on the underlying app GUI.

VI. USER STUDY

We conducted a preliminary lab usability study. Participants were asked to use APPINITE to specify data descriptions in 20 example scenarios. The purpose of the study was to evaluate the usability of APPINITE on combining natural language instructions and demonstrations.

A. Participants

We recruited 6 participants (1 woman and 5 men, average age = 26.2) at Carnegie Mellon University. All but one of the participants were graduate students in technical fields. All participants were active smartphone users, but none had used APPINITE prior to the study. Each participant was paid \$15 for an 1-hour user study session.

Although the programming literacy of our participants is not representative of our target users, this was not a goal of this study. The primary goal was to evaluate the usability of our interaction design on combining natural language instructions and demonstrations. The demonstration part of this usability study was based on SUGILITE’s [2], which found no significant difference in PBD task performances among groups with different programming expertise. Our formative study (Section III) showed that non-programmers were able to provide adequate natural language instructions from which APPINITE can generate generalizable data descriptions.

B. Tasks

From the top free apps in Google Play, we picked 8 sample apps (OpenTable, Kayak, Amtrak, Walmart, Hotel Tonight, Fly Delta, Trulia and Airbnb) where we identified data description challenges. Within these apps, we designed 20 scenarios. Each scenario required the participant to demonstrate choosing an UI object from a collection of options. All the target UI objects had multiple possible and reasonable data descriptions where the correct ones (that reflect user intentions) could not be inferred from demonstrations alone, or using heuristic rules without semantic understanding of the context. The tasks required participants to specify data descriptions using APPINITE. For each scenario, the intended feature for the data description was communicated to the participant by pointing at the feature on the screen. Spoken instructions from the experimenter were minimized, and carefully chosen to avoid biasing what the participant would say. Four out of the 20 scenarios were set up in a

way that multi-turn conversations for disambiguation (e.g., Fig. 1c and Fig. 1d) were needed. The chosen sample scenarios used a variety of different domains, GUI layouts, data description features, and types of expressions in target queries (i.e. joins, conjunctions and superlatives).

C. Procedure

After obtaining consent, the experimenter first gave each participant a 5-minute tutorial on how to use APPINITE. During the tutorial, the experimenter showed the supplemental video figure as an example to explain APPINITE’s features.

Following the tutorial, each participant was shown the 8 sample apps in random order on a Nexus 5X phone. For each scenario within each app, the experimenter navigated the app to the designated state before handing the phone to the participant. The experimenter pointed to the UI object which the participant should demonstrate clicking on, and pointed to the on-screen feature which the participant should use for verbally describing the intention. For each scenario, the participant was asked to demonstrate the action, provide the natural language description of intention, and complete the disambiguation conversation if prompted by APPINITE. The participant could retry if the speech recognition was incorrect, and try a different instruction if the parsing result was different from expected. APPINITE recorded participants’ instructions as well as the corresponding UI snapshot graphs, the demonstrations, and the parsing results.

After completing the tasks, each participant was asked to complete a short survey, where they rated statements about their experience with APPINITE on a 7-point Likert scale. The experimenter also had a short informal interview with each participant to solicit their comments and feedback.

D. Results

Overall, our participants had a good task completion rate. Among all 120 scenario instances across the 6 participants, 106 (87%) were successful in producing the intended target data description query on the first try. Note that we did not count retries caused by speech recognition errors, as it was not a focus of this study. Failed scenarios were all caused by incorrect or failed parsing of natural language instructions, which can be fixed by (1) having bigger training datasets with better coverage for words and expressions users may use in instructions, and (2) enabling better semantic understanding of GUIs (details in Section VII). Participants successfully completed all initially failed scenarios in retries by rewording their verbal instructions after being prompted by APPINITE. Among all the 120 scenario instances, 24 instances required participants to have multi-turn conversations for disambiguation. 22 of these 24 (92%) were successful on the first try, and the rest were fixed by rewording.

In our survey on a 7-point Likert scale from “strongly disagree” to “strongly agree”, our 6 participants found APPINITE “helpful in programming by demonstration” (mean=7), “allowed them to express their intentions naturally” (mean=6.8, $\sigma=0.4$), and “easy to use” (mean=7). They also agreed that “the multi-modal interface of APPINITE is helpful” (mean=6.8, $\sigma=0.4$), “the real-time visualization is helpful for disambiguation” (mean=6.7, $\sigma=0.5$), and “the error messages are helpful” (mean=6.8, $\sigma=0.4$).

VII. DISCUSSION AND FUTURE WORK

The study results suggested that APPINITE has good usability, and also that it has adequate performance for generating correct formal executable data description queries from demonstrations and natural language instructions in the sample scenarios. As the next step, we plan to run offline performance evaluations for the UI snapshot graph extractor and the natural language instruction parser, and in-situ field studies to evaluate APPINITE’s usage and performance for organic tasks in real-world settings.

Participants praised APPINITE’s usefulness and ease of use. A participant reported that he found sample tasks very useful to have done by an intelligent agent. Participants also noted that without APPINITE, it would be almost impossible for end users without programming expertise to create automation scripts for these tasks, and it would also take considerable effort for experienced programmers to do so.

Our results illustrate the effectiveness of combining two input modalities, each with its own different of ambiguities, to more accurately infer user’s intentions in EUD. A major challenge in EUD is that end users are unable to precisely specify their intended behaviors in formal language. Thus, easier-to-use but ambiguous alternative programming techniques like PBD and natural language programming are adopted. Our results suggest that end users can effectively clarify their intentions in a complementary technique with adequate guidance from the system when the initial input was ambiguous. Further research is needed on how users naturally select modalities in multi-modal environments, and on how interfaces can support more fluid transition between modalities.

Another insight is to leverage the GUI as a shared grounding for EUD. By asking users to describe intentions in natural language referring to GUI contents, our tool constrains the scope of instructions to a limited space, making semantic parsing feasible. Since users are already familiar with app GUIs, they do not have to learn new symbols or mechanisms as in scripting or visual languages. The knowledge graph extraction further provides users with greater expressive power by abstracting higher-level semantics from platform-specific implementations, enabling users to talk about semantic relations for the items such as “the cheapest restaurant” and “the score for Minnesota.”

While APPINITE has already offered some semantic-based features to provide greater expressiveness than existing end user PBD task automation tools, participants were hoping for more powerful support to enable them to naturally express more complicated logic in a more flexible way. To achieve this, we plan to improve APPINITE in the following areas:

A. Learning Conceptual Knowledge

We plan to leverage recent advances in natural language processing (e.g., [53]) to enable APPINITE to learn new concepts from users through verbal instructions. More specifically, we want to support users to add new relations into UI snapshot graphs through conversations. For example, for the interface shown in Fig. 1a, users can currently say, “*the restaurant that is 804 feet away*” (corresponds to the `hasText` relation) or “*the closest restaurant*” (corresponds to the `containsDistance` relation), but not “*restaurants within walking distance*” as APPINITE does not yet know the concept of “walking distance.” We plan to enable future versions of APPINITE to ask users to

explain unknown (and possibly personalized) concepts. For this example, a user may say “*Walking distance means less than half a mile*”, from which APPINITE can define a relation extractor for the `isWalkingDistance` modifier for existing objects with the `containsDistance` relation, and subsequently allow use of the new concept “walking distance” in future instructions.

B. Computation in Natural Language Instructions

Currently in APPINITE, users have a limited capability of specifying computations and comparisons in natural language instructions. For example, for the interface shown in Fig. 2b, users cannot use expressions like “*flights that are cheaper than \$700*” or “*if the flight is shorter than 4 hours*” in specifying data descriptions, although the UI snapshot graph already contains the prices and the durations for all flights. Furthermore, users are not able to create control structures (e.g., conditionals, iterations, triggers) which would require computations and comparisons. To address this issue, we plan to leverage prior work on natural language programming [32], and more importantly, how non-programmers can naturally describe computations, control structures and logic in solutions to programming problems [54] to extend our parser so that it can understand naturally expressed computations and the corresponding control structures. However, even with advanced semantic parsing and natural language processing techniques, GUI demonstrations will still remain essential for grounding users’ natural language inputs and resolving ambiguities in the natural language.

C. Better Semantic Understanding of GUIs

Future versions of APPINITE can benefit from having better semantic understanding of GUIs. Some understanding can be acquired from user instructions, while others can come from existing resources. As discussed previously, the format of our UI snapshot graph allows easy integration with existing knowledge bases, which enables APPINITE to understand the semantics of entities (e.g., JetBlue, Delta and American are all instances of `airlines` for the interface in Fig. 2b). This integration can allow APPINITE to have more accurate instruction parsing, and to ask more specific questions in follow-up conversations.

GUI layouts can also be better leveraged to extract semantics. So far, we have only used the inter-object binary geometric relations such as `above` and `nextTo` to represent possible semantic relations between individual UI objects, but not the overall layout. Prior research suggests that app GUI designs often follow common design patterns, where the layout can suggest its functionality [55]. Also, for graphics in GUIs, especially for those without developer-provided accessibility labels, we can use runtime annotation techniques [56] to annotate their meanings. Visual features in GUIs can also be used in data descriptions, as discussed in [6], [57], [58].

VIII. CONCLUSION

Natural language instruction is a natural and expressive medium for users to specify their intentions and can provide useful complementary information about user intentions when used in conjunction with other EUD approaches, such as PBD. APPINITE combines natural language instructions with demonstrations to provide end users with greater expressive power to create more generalized GUI automation scripts, while retaining usability, transparency and understandability.

REFERENCES

- [1] T. J.-J. Li, I. Labutov, B. A. Myers, A. Azaria, A. I. Rudnick, and T. M. Mitchell, “An End User Development Approach for Failure Handling in Goal-oriented Conversational Agents,” in *Studies in Conversational UX Design*, Springer, 2018.
- [2] T. J.-J. Li, A. Azaria, and B. A. Myers, “SUGILITE: Creating Multi-modal Smartphone Automation by Demonstration,” in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2017, pp. 6038–6049.
- [3] J. Allen *et al.*, “Plow: A collaborative task learning agent,” in *Proceedings of the National Conference on Artificial Intelligence*, 2007, vol. 22, p. 1514.
- [4] G. Leshed, E. M. Haber, T. Matthews, and T. Lau, “CoScripter: Automating & Sharing How-to Knowledge in the Enterprise,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2008, pp. 1719–1728.
- [5] M. Bolin, M. Webber, P. Rha, T. Wilson, and R. C. Miller, “Automation and customization of rendered web pages,” in *Proceedings of the 18th annual ACM symposium on User interface software and technology*, 2005, pp. 163–172.
- [6] 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*, New York, NY, USA, 2009, pp. 183–192.
- [7] “Automate: everyday automation for Android. LlamaLab.” [Online]. Available: <http://llamalab.com/automate/>. [Accessed: 11-Sep-2016].
- [8] S. K. Kuttal, A. Sarma, and G. Rothermel, “History repeats itself more easily when you log it: Versioning for mashups,” in *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2011, pp. 69–72.
- [9] M. Pruitt, *Yahoo! Pipes*, First. O’Reilly, 2007.
- [10] A. Cypher and D. C. Halbert, *Watch what I do: programming by demonstration*. MIT press, 1993.
- [11] H. Lieberman, *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.
- [12] D. C. Halbert, “SmallStar: programming by demonstration in the desktop metaphor,” in *Watch what I do*, 1993, pp. 103–123.
- [13] B. A. Myers and R. McDaniel, “Sometimes you need a little intelligence, sometimes you need a lot,” *Your Wish My Command Program. Ex. San Franc. CA Morgan Kaufmann Publ.*, pp. 45–60, 2001.
- [14] B. A. Myers, “Scripting graphical applications by demonstration,” in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 1998, pp. 534–541.
- [15] B. A. Myers, “Peridot: creating user interfaces by demonstration,” in *Watch what I do*, 1993, pp. 125–153.
- [16] S. Gulwani, “Automating String Processing in Spreadsheets Using Input-output Examples,” in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, NY, USA, 2011, pp. 317–330.
- [17] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld, “Programming by Demonstration Using Version Space Algebra,” *Mach Learn*, vol. 53, no. 1–2, pp. 111–156, Oct. 2003.
- [18] T. J.-J. Li and O. Riva, “KITE: Building conversational bots from mobile apps,” in *Proceedings of the 16th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys 2018)*, 2018.
- [19] R. G. McDaniel and B. A. Myers, “Getting More out of Programming-by-demonstration,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 1999, pp. 442–449.
- [20] A. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. Kalai, “A Machine Learning Framework for Programming by Example,” presented at the Proceedings of the 30th International Conference on Machine Learning (ICML-13), 2013, pp. 187–195.
- [21] T. Lau, “Why Programming-By-Demonstration Systems Fail: Lessons Learned for Usable AI,” *AI Mag.*, vol. 30, no. 4, pp. 65–67, Oct. 2009.
- [22] T. Y. Lee, C. Dugan, and B. B. Bederson, “Towards Understanding Human Mistakes of Programming by Example: An Online User Study,” in *Proceedings of the 22Nd International Conference on Intelligent User Interfaces*, New York, NY, USA, 2017, pp. 257–261.
- [23] S. Oviatt, “Ten Myths of Multimodal Interaction,” *Commun ACM*, vol. 42, no. 11, pp. 74–81, Nov. 1999.
- [24] R. A. Bolt, “‘Put-that-there’: Voice and Gesture at the Graphics Interface,” in *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, USA, 1980, pp. 262–270.
- [25] R. Marin, P. J. Sanz, P. Nebot, and R. Wirz, “A multimodal interface to control a robot arm via the web: a case study on remote programming,” *IEEE Trans. Ind. Electron.*, vol. 52, no. 6, pp. 1506–1520, Dec. 2005.
- [26] S. Iba, C. J. J. Paredis, and P. K. Khosla, “Interactive Multimodal Robot Programming,” *Int. J. Robot. Res.*, vol. 24, no. 1, pp. 83–104, Jan. 2005.
- [27] S. Oviatt, “Mutual disambiguation of recognition errors in a multi-model architecture,” in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, 1999, pp. 576–583.
- [28] E. Adar, M. Dontcheva, and G. Laput, “CommandSpace: Modeling the Relationships Between Tasks, Descriptions and Features,” in *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, New York, NY, USA, 2014, pp. 167–176.
- [29] T. Kasturi *et al.*, “The Cohort and Speechify Libraries for Rapid Construction of Speech Enabled Applications for Android,” in *Proceedings of the 16th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, 2015, pp. 441–443.
- [30] G. P. Laput *et al.*, “PixelTone: A Multimodal Interface for Image Editing,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2013, pp. 2185–2194.
- [31] A. Azaria, J. Krishnamurthy, and T. M. Mitchell, “Instructable Intelligent Personal Agent,” in *Proc. The 30th AAAI Conference on Artificial Intelligence (AAAI)*, 2016, vol. 4.
- [32] A. W. Biermann, “Natural Language Programming,” in *Computer Program Synthesis Methodologies*, Springer, Dordrecht, 1983, pp. 335–368.
- [33] D. Price, E. Riloff, J. Zachary, and B. Harvey, “NaturalJava: A Natural Language Interface for Programming in Java,” in *Proceedings of the 5th International Conference on Intelligent User Interfaces*, New York, NY, USA, 2000, pp. 207–211.
- [34] A. Begel and S. L. Graham, “Spoken programs,” in *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC’05)*, 2005, pp. 99–106.
- [35] H. Lieberman and H. Liu, “Feasibility studies for programming in natural language,” in *End User Development*, Springer, 2006, pp. 459–473.
- [36] D. L. Chen and R. J. Mooney, “Learning to Interpret Natural Language Navigation Instructions from Observations,” in *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, San Francisco, California, 2011, pp. 859–865.
- [37] J. Thomason, S. Zhang, R. Mooney, and P. Stone, “Learning to Interpret Natural Language Commands Through Human-robot Dialog,” in *Proceedings of the 24th International Conference on Artificial Intelligence*, Buenos Aires, Argentina, 2015, pp. 1923–1929.
- [38] V. Le, S. Gulwani, and Z. Su, “SmartSynth: Synthesizing Smartphone Automation Scripts from Natural Language,” in *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, New York, NY, USA, 2013, pp. 193–206.
- [39] C. Huff and D. Tingley, “Who are these people? Evaluating the demographic characteristics and political preferences of MTurk survey respondents,” *Res. Polit.*, vol. 2, no. 3, p. 2053168015604648, 2015.
- [40] T. J.-J. Li, Y. Li, F. Chen, and B. A. Myers, “Programming IoT Devices by Demonstration Using Mobile Apps,” in *End-User Development*, Cham, 2017, pp. 3–17.
- [41] Google, “AccessibilityWindowInfo | Android Developers.” [Online]. Available: <https://developer.android.com/reference/android/view/accessibility/AccessibilityWindowInfo.html>. [Accessed: 23-Apr-2018].
- [42] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, “Dbpedia: A nucleus for a web of open data,” *Semantic Web*, pp. 722–735, 2007.
- [43] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, “Freebase: a collaboratively created graph database for structuring human knowledge,” in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 1247–1250.
- [44] D. Vrandečić and M. Krötzsch, “Wikidata: a free collaborative knowledgebase,” *Commun. ACM*, vol. 57, no. 10, pp. 78–85, 2014.

- [45] S. Sen, T. J.-J. Li, WikiBrain Team, and B. Hecht, "WikiBrain: Democratizing computation on Wikipedia," in *Proceedings of the 10th International Symposium on Open Collaboration (WikiSym + OpenSym 2014)*, 2014.
- [46] T. J.-J. Li, S. Sen, and B. Hecht, "Leveraging Advances in Natural Language Processing to Better Understand Tobler's First Law of Geography," in *Proceedings of the 22Nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, New York, NY, USA, 2014, pp. 513–516.
- [47] P. Liang, M. I. Jordan, and D. Klein, "Learning dependency-based compositional semantics," *Comput. Linguist.*, vol. 39, no. 2, pp. 389–446, 2013.
- [48] P. Pasupat and P. Liang, "Compositional Semantic Parsing on Semi-Structured Tables," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*, 2015.
- [49] J. Berant, A. Chou, R. Frostig, and P. Liang, "Semantic parsing on freebase from question-answer pairs," in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, 2013, pp. 1533–1544.
- [50] M. Toomim, S. M. Drucker, M. Dontcheva, A. Rahimi, B. Thomson, and J. A. Landay, "Attaching UI Enhancements to Websites with End Users," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2009, pp. 1859–1868.
- [51] J. R. Eagan, M. Beaudouin-Lafon, and W. E. Mackay, "Cracking the Cocoa Nut: User Interface Programming at Runtime," in *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, New York, NY, USA, 2011, pp. 225–234.
- [52] X. Zhang, A. S. Ross, A. Caspi, J. Fogarty, and J. O. Wobbrock, "Interaction Proxies for Runtime Repair and Enhancement of Mobile Application Accessibility," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2017, pp. 6024–6037.
- [53] S. Srivastava, I. Labutov, and T. Mitchell, "Joint concept learning and semantic parsing from natural language explanations," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, 2017, pp. 1527–1536.
- [54] J. F. Pane, B. A. Myers, and others, "Studying the language and structure in non-programmers' solutions to programming problems," *Int. J. Hum.-Comput. Stud.*, vol. 54, no. 2, pp. 237–264, 2001.
- [55] B. Deka, Z. Huang, and R. Kumar, "ERICA: Interaction Mining Mobile Apps," in *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, New York, NY, USA, 2016, pp. 767–776.
- [56] X. Zhang, A. S. Ross, and J. Fogarty, "Robust Annotation of Mobile Application Interfaces in Methods for Accessibility Repair and Enhancement," in *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, 2018.
- [57] T. Intharah, D. Turmukhambetov, and G. J. Brostow, "Help, It Looks Confusing: GUI Task Automation Through Demonstration and Follow-up Questions," in *Proceedings of the 22Nd International Conference on Intelligent User Interfaces*, New York, NY, USA, 2017, pp. 233–243.
- [58] M. Dixon and J. Fogarty, "Prefab: Implementing Advanced Behaviors Using Pixel-based Reverse Engineering of Interface Structure," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2010, pp. 1525–1534.

The Impact of Culture on Learner Behavior in Visual Debuggers

Kyle Thayer

*Paul G. Allen School of
Computer Science & Engineering
University of Washington
Seattle, WA, USA
kthayer@cs.washington.edu*

Philip J. Guo

*Dept. of Cognitive Science
UC San Diego
La Jolla, CA, USA
pg@ucsd.edu*

Katharina Reinecke

*Paul G. Allen School of
Computer Science & Engineering
University of Washington
Seattle, WA, USA
reinecke@cs.washington.edu*

Abstract—People around the world are learning to code using online resources. However, research has found that these learners might not gain equal benefit from such resources, in particular because culture may affect how people learn from and use online resources. We therefore expect to see cultural differences in how people use and benefit from visual debuggers. We investigated the use of one popular online debugger which allows users to execute Python code and navigate bidirectionally through the execution using forward-steps and back-steps. We examined behavioral logs of 78,369 users from 69 countries and conducted an experiment with 522 participants from 82 countries. We found that people from countries that tend to prefer self-directed learning (such as those from countries with a low Power Distance, which tend to be less hierarchical than others) used about twice as many back-steps. We also found that for individuals whose values aligned with instructor-directed learning (those who scored high on a “Conservation” scale), back-steps were associated with less debugging success.

Index Terms—program visualization, cross-cultural studies, non-linear learning

I. INTRODUCTION

People from over 180 countries are learning computer programming from online resources [1], [2]. Though interest in learning to code is widespread internationally, the dominant programming education tools and MOOC platforms that teach such skills (e.g., edX, Coursera, Udacity, Codecademy, Code.org) were developed by people in the United States. This creates a risk that designers may have unconsciously embedded their cultural values into these platforms, making them less suitable for people in other countries. Indeed, researchers have raised concerns about whether MOOC and programming education resources are optimized for, and primarily benefit, those who come from more privileged Western-centric backgrounds [3]–[7]. One of the reasons for these concerns is that a country’s national culture has been found to influence how people learn [7]–[9]. For example, prior work found that countries with a high power distance—an indicator of strong hierarchies, such as found in India and China [10]—often employ instructor-directed education [8]. Students who grow up in these countries are thought to be more used to and may prefer step-by-step instructions and more linear navigation [2], [11]. In contrast, students from

countries with a low power distance, such as the U.S. and Denmark, exhibit more self-directed learning and might prefer more self-guided and less linear navigation. This phenomenon has been seen in MOOCs, where students from low power distance countries navigate the content more non-linearly (i.e., jumping back and forth between different sections) than those from high power distance countries [2].

Our main question in this work is whether such differences can also be found among users of visual debuggers. *Do people from different cultures use and benefit from visual debuggers in distinct ways?* and more specifically, *Does a propensity for self-directed learning explain some of these differences?* If yes, this would suggest that visual debuggers might have to be adjusted to optimally support users from different cultural backgrounds. It would also indicate that cultural differences in behavior prevail even within a relatively homogeneous group of users who seek out online debugging tools to learn programming.

As a first concrete step toward investigating these questions, we evaluate how learners from over 60 countries engage with a specific feature within Python Tutor, a popular visualization-based online debugger often used in conjunction with programming tutorials [12]. Central to Python Tutor is the beginner-friendly feature of bidirectional navigation of code executions [13]. This feature allows users to jump both to earlier steps (“back-steps”) and later steps (“forward-steps”) of the code execution while running a piece of code (Figure 1). Given the prior work on the influence of culture on the level of self-directed learning [2], [8], we hypothesize cultural levels of self-directed learning will correlate with navigation by back-steps in Python Tutor. We conducted two quantitative studies to probe this hypothesis, using two proxy measures for instructor-directed learning: *Power Distance Indicator* (PDI, a measure of how hierarchical a country is) and *Conservation* (a measure of how much an individual values tradition, conformity, and security) [10], [14].

In our first study, we analyzed behavioral log data from 78,369 users of Python Tutor over six months. We found significant differences between countries in how many back-steps their users took. In particular, users from low and medium PDI countries, such as Israel, Germany or the US, took *more* back-

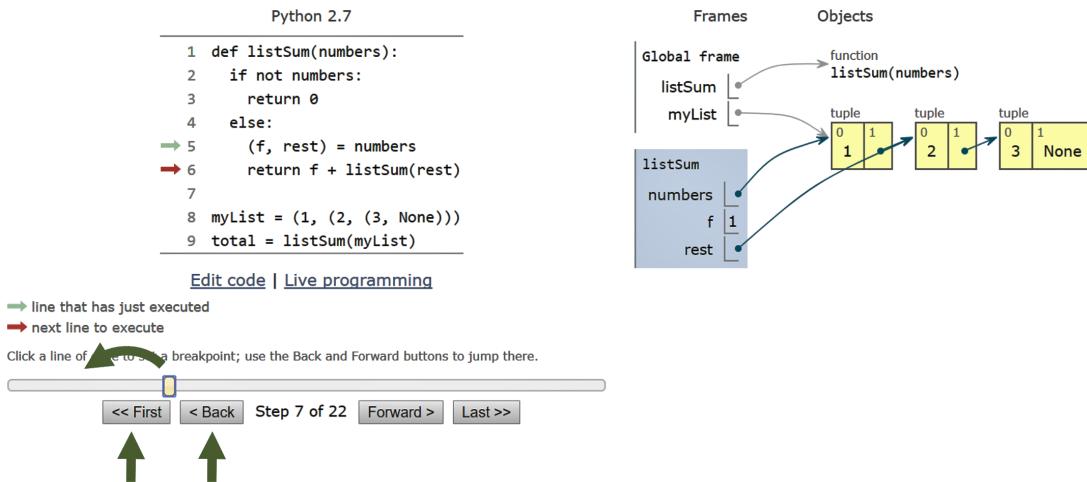


Fig. 1. Python Tutor [12] lets learners navigate through example programs and visually debug their code. The large green arrows annotate the three ways of performing *back-steps* to jump back to earlier execution points.

steps when following code executions on Python Tutor than those from high PDI countries, such as India, China, or Russia. People in the most egalitarian countries (with low PDI and self-directed education, where students are often encouraged to find their own way to solve problems) took about twice as many backward steps through code executions than those from the most hierarchical countries (with high PDI and instructor-directed education).

Since individuals' culture and their propensity for self-directed learning varies within countries, we conducted a second study to investigate the relationship between back-steps and self-directed learning at an individual level. For this study we recruited 522 participants to perform a debugging activity on Python Tutor and asked them to answer a questionnaire to assess Conservation as an individual measure of cultural values. Participants' Conservation scores were marginally correlated with the number of back-steps. We did not find a correlation between PDI and back-steps as in the first study, but we again found differences between some countries in the use of back-steps. In addition, contrary to our expectations, back-steps correlated negatively with debugging success, but this effect varied with Conservation score.

Altogether, our results show that people do not uniformly use visual debuggers and do not equally benefit from certain functionalities. The national cultural dimension Power Distance and individual's Conservation score can predict some of these differences. Our study makes the following new contributions to the research area of cross-cultural influences on learning technologies:

- 1) We contribute the first studies of the effects of learners' culture on their use of visual debuggers. We found differences in how learners from various countries use the back-step feature in Python Tutor, a widely-used online debugger commonly used with tutorials. Our studies suggest that these differences can be partially explained by users' level of self-directed learning as measured by Power Distance and Conservation.
- 2) Our results showed that for individuals whose values

aligned with instructor-directed learning (high Conservation), back-steps were associated with less debugging success.

- 3) Our findings also point to how users from different cultures may benefit from different presentations of non-linear navigation features in online programming education tools. The Power Distance Index, which can be easily derived from IP addresses without any additional input from the user, can be used as a rough approximation of culture when doing these adaptations.

II. THE PYTHON TUTOR WEBSITE

The Python Tutor website [12], [15] is an open-source code visualization system that allows learners to edit and debug code directly in their web browser. The system has two views: 1) a code editor view allows users to write code and press a button to run their code, which opens 2) a run-time state visualization view (Figure 1) that lets users debug their code by allowing them to navigate all of the steps of program execution, both forwards and backwards, using a slider or buttons. At each step the user sees all variables, values, stack, heap, and textual output at that point in execution.

The Python Tutor website hosts a set of basic programming examples where learners can execute the example code and step through visualizations of its run-time state. In addition, many users copy and paste code from other websites (e.g., MOOCs, blogs) into Python Tutor's code editor to understand and debug it using the visualizations of its run-time state.

III. BACKGROUND AND HYPOTHESIS DEVELOPMENT

We developed hypotheses based on prior work on cultural measures and how those relate to people's behavior.

A. Culture and Back-Steps

According to Hofstede, culture describes a shared "programming of the mind" [16], which results in groups of people having shared values and preferences [17]. Culture is not easily defined; in fact, researchers debate what exactly it describes and what influences culture has. Culture cannot be constrained

to country borders [18], but people from the same country can still share a national culture and might often adhere to certain behavioral trends [10], [16].

Grappling with the issue of trying to define cultures, researchers have attempted to quantify differences between cultures, while acknowledging that any differences can only describe trends and are not going to generalize to all members of a specific culture. Two notable efforts to measure culture are by Hofstede [10] and Schwartz [14]. Hofstede's cultural dimensions measures culture at a national level, while Schwartz found a universal structure to the value trade-offs individual people make, holding true across different countries.

From Hofstede's cultural dimensions, his Power Distance Index (PDI) is the most relevant to the aspect of self-directed learning that we are investigating. PDI measures “the extent to which the less powerful persons in a society accept inequality in power and consider it normal” [8]. Societies with a higher PDI (e.g., India or China) tend to have more “teacher-centered education (premium on order),” where the “students expect the teacher to outline paths to follow,” the “teacher is never contradicted nor publicly criticized,” and the “effectiveness of learning [is] related to the excellence of the teacher” [8]. Learning in these high PDI environments is centered on the authority of the instructor and thus is instructor-directed. In contrast, societies with a lower PDI (e.g., the US or many Western European countries) have more “student-centered education (premium on initiative),” where the “teacher expects students to find their own path,” the “students [are] allowed to contradict or criticize the teacher,” and the “effectiveness of learning [is] related to amount of two-way communication in class” [8]. Education in these low PDI environments is centered on each student’s individual authority and thus is more self-directed. Lower PDI countries also tend to have more resources available to put toward education: they have smaller class sizes¹ and higher GDP per capita² (see also [2]).

Such differences in day-to-day education likely translate into people’s learning behavior online, even after they have finished school. Indeed, low student-teacher ratios in a country (which is associated with self-directed learning [8]), was found to correlate with students making more “backjumps” in MOOCs, where they navigate to earlier course content [2]. For those in high PDI countries, prior research has suggested providing linear navigation, reducing navigation choices and providing support through wizard interfaces [11], [21].

Inspired by this line of work, we expect programming learners to view Python Tutor as a computerized “instructor” and thus that learners from high PDI countries (with more instructor-directed learning) will view individual steps in the code visualization as the canonical intended path offered by Python Tutor. Since Python Tutor gives no explicit instructions to step either forward or backwards, we expect these users to

assume any steps, from the first to last execution step, were intended to be followed forward in a linear order.

Conversely, we expect users from low PDI countries (with more self-directed learning) assume that Python Tutor (as a computerized “instructor”) gives them a space of options to explore, and that they will see the execution steps as intended to be navigated in whatever order best fits their needs. Thus, we hypothesize that users from higher PDI countries will take fewer back-steps, and users from lower PDI countries will take more back-steps:

[H.1] *PDI negatively correlates with the number of back-steps that users take in Python Tutor’s code visualizations.*

Since national cultural measures (like PDI) do not take individual differences into account, we also wanted to investigate how personal values of self-directed learning relate to using back-steps. We wanted to use a validated individual cultural measure for this, so we chose the one most relevant to the aspects self-directed learning that we are investigating: Conservation vs. Openness-to-change from Schwartz’s universal values work [14]. Schwartz’s values have been shown to correlate with decision making, political views, and observed behavior [22]–[24], with those who score higher on openness-to-change being more willing to follow their own interests in unpredictable directions [14]. Since Conservation can be an appropriate proxy measure of self-directed learning like PDI, we assume it will relate to how back-steps are used. Our second hypothesis is therefore:

[H.2] *Conservation score will negatively correlate with back-steps in Python Tutor code visualizations.*

B. The Efficacy of Back-Steps for Code Debugging

The closest related technical systems to Python Tutor are backwards-in-time debuggers that allow a user to navigate from a given point in code execution back to earlier execution steps. This feature allows programmers to find the causes of bugs without guessing where to set breakpoints [25]–[27]. Most research on backwards-in-time debuggers has focused on the debugging techniques and technical implementations [25]–[31]. To our knowledge, there have been no prior studies of how users’ national culture and personal values affect their use of code debuggers.

Since backwards-in-time debuggers were specifically built to help with debugging (and one study on a specific variant found them to be beneficial [31]), we hypothesize that back-stepping will generally correlate with more debugging success (in terms of how many tests the modified user code passes):

[H.3] *Back-steps in code visualizations will correlate with debugging success.*

We also hypothesize that self-directed learners (low Conservation) will have had more experience choosing their own path and be more comfortable breaking from linear orders. These learners might be more prepared and able to make effective use of back-steps. Therefore we expect the use of back-steps by self-directed learners to more likely result in debugging success than the back-steps of instructor-directed learners. Thus, we expect an interaction effect: Self-directed learners

¹PDI is correlated with student-teacher ratio (using data provided by [19]), $r(47) = .37, p < .01$

²PDI is negatively correlated with GDP per capita (using data from [20]), $r(65) = -.62, p < .0001$.

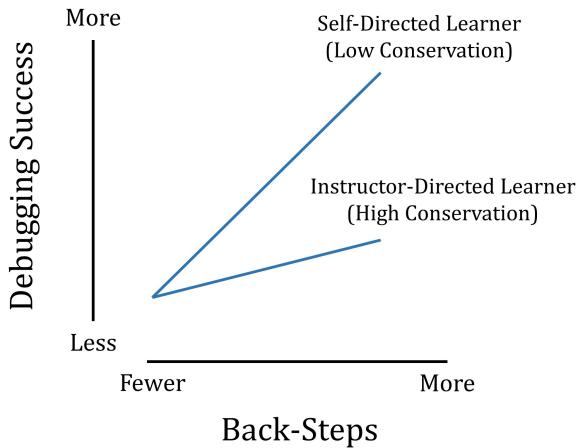


Fig. 2. Hypothesis H.4: We expect an interaction effect between self-directed learning, back-steps, and debugging success. We expect debugging success to positively correlate with back-steps for all learners. But we expect self-directed learners benefit more from any back-steps they take, and thus will have a larger positive correlation between back-steps and debugging success.

(low Conservation) benefit more from back-steps (thus a larger correlation between back-steps and debugging success) than those participants who are used to instructor-directed learning (see Figure 2):

[H.4] Conservation interacts with the correlation between back-steps and debugging success. For lower Conservation learners, the correlation will be stronger, while for higher Conservation learners, the correlation will be weaker.

IV. STUDY 1: PYTHON TUTOR BEHAVIORAL LOG ANALYSIS

For our first study, we examined behavioral log data from Python Tutor in order to test if back-steps are negatively correlated with Power Distance (H.1).

A. Methods

To test our hypothesis, we retrieved six months of behavioral log data from Python Tutor and supplemented the dataset with the Power Distance scores for each user's country. The dataset comprised the following:

- User events, allowing us to calculate features such as back-steps, forward-steps, time spent, and code length;
- 78,369 unique user IDs (UUIDs);
- Browser sessions, allowing us to track user events across multiple code visualizations in a session;
- User country, deduced from their IP address using the GeoLite2 Free database [32];
- The Power Distance Index for each user based on their country and Hofstede's official country PDI scores [33].

Python Tutor does not ask users to sign up or provide any demographic information, so we were unable to control for possible effects of demographics.

1) Users: Our dataset included 147,847 users who visited the Python Tutor website from 166 countries. We removed users who did not use code visualizations, who did not take any steps in a code visualization, or whose country was not part of Hofstede's study and therefore could not be linked to PDI.

The final dataset included 1,236,863 code visualizations run by 78,369 unique users from 69 countries. The US accounted for 32% of the data, India for 7.8%, and the UK for 5.3%. The average PDI for users was 50.4 ($SD = 17.8$), which is roughly half of the maximum possible PDI value of 120.

2) Analysis: We conducted a series of mixed-model analysis of variances on code execution visualizations, with the back-step count as the dependent variable.³. We modelled PDI as an independent factor. Because we wanted to do our analysis on individual code execution visualizations and a large numbers of users who interacted with multiple code visualizations, we modelled user ID as a random factor. Since Python Tutor users' exact tasks were unknown to us (users were free to follow any example on the site or copy and paste in any code from elsewhere), we controlled for 13 additional variables (see Table I) that measured either engagement (such as time spent and forward-steps) or code properties (such as length of code and number of exceptions thrown when running the code). To further understand user's tasks and the code they were running, we also examined all code executions for 20 random browser sessions in four different countries with at least 10,000 code executions: two with high PDI and few average back-steps (Russia and India), and two with low PDI and more average back-steps (Israel and Australia).

B. Results

Our linear regression confirmed H.1: Power Distance was negatively correlated with the number of back-steps in a code execution visualization ($F_{(1,47489)} = 84, p < .0001, \beta = -.052, t\text{-value} = -9.1$) (Figure 3). For example, picking the most and fewest average back-steps per code execution for countries with at least 10,000 code executions, we found significant differences between Israel ($M = 1.7, SD = 4.2, PDI = 13$) and India ($M = 0.38, SD = 1.7, PDI = 77$); $t_{(-49)} = 26206, p < .0001$.⁴ For the other variables, higher engagement with the Python Tutor tool correlated with more back-steps; most notably with forward-steps taken ($F_{(1,1235156)} = 178878, p < .0001, \beta = 1.2, t\text{-value} = 423$), time spent in the visualization ($F_{(1,1191571)} = 7090, p < .0001, \beta = 0.23, t\text{-value} = 84$) and length of the code (in characters) ($F_{(1,881703)} = 2201, p < .0001, \beta = 0.15, t\text{-value} = 46$). The full regression table is shown in Table I.

Examining all code executions for randomly selected browser sessions allowed us to see how users were modifying and executing code. Our observations included that code and apparent task varied greatly within countries; in addition, we saw few differences between the countries. Code being edited ranged widely, from apparent tests of how python list functions worked to sorting functions to dice rolling games to string processing, all with no apparent difference between the high

³While back-steps were not normally distributed, linear regressions are thought to be robust to outliers and other violations of assumptions for large samples such as ours [34]

⁴While these are the medians of skewed data, the large sample size still allow for comparison. [34]

TABLE I

ANALYSIS OF VARIANCE RESULTS FOR ALL FACTORS IN THE REGRESSION MODEL FOR *back-steps* (STUDY 1), EXCLUDING UserID, WHICH WAS A RANDOM FACTOR. FACTORS CAN BE AT ONE OF THREE SCOPES: *User* IS A VALUE THAT IS CONSTANT FOR A USER ACROSS ALL TIME; *Execution* IS A VARIABLE SCOPED TO A SINGLE CODE VISUALIZATION EXECUTION; *Session* IS A VARIABLE THAT IS SHARED ACROSS A BROWSER SESSION BY A USER WHERE THE USER MAY HAVE RUN MULTIPLE CODE VISUALIZATION EXECUTIONS. THIS MODEL SHOWS THAT PDI NEGATIVELY CORRELATED WITH *back-steps*, AND THAT MANY FACTORS MEASURING ENGAGEMENT (SUCH AS *Time*) WERE POSITIVELY CORRELATED WITH BACK-STEPS. WE ALSO INCLUDED THE COEFFICIENTS FROM THE LINEAR MODEL TO ALLOW COMPARISON (ALL NON-BOOLEAN INDEPENDENT VARIABLES WERE NORMALIZED). MARGINAL $R^2 = .18$ (VARIANCE EXPLAINED BY FIXED FACTORS), CONDITIONAL $R^2 = .28$ (THE VARIANCE EXPLAINED BY FIXED AND RANDOM FACTORS COMBINED).

Scope	Variable	Coeff.	df	F	p-value
User	PDI	-0.052	1	84	< .0001 ***
Execution	Time	0.23	1	7090	< .0001 ***
Execution	# steps available	0.024	1	55	< .0001 ***
Execution	# of forward-steps	1.2	1	178878	< .0001 ***
Execution	Length of code (# chars)	0.15	1	2201	< .0001 ***
Execution	Edit-dist. from previous execution	-0.041	1	240	< .0001 ***
Execution	Execution number in session	0.0052	1	1	= .22 (n.s.)
Execution	Was code just edited?	-0.0028	1	15	< .0001 ***
Execution	# of function calls	0.0060	1	0	= .05 *
Execution	# of exceptions	0.063	1	598	< .0001 ***
Session	Total forward-steps	-0.017	1	13	= .0004 ***
Session	Total edit-distance	0.015	1	16	< .0001 ***
Session	# of executions	-0.030	1	23	< .0001 ***
Session	Was code ever edited?	0.067	1	0	= .66 (n.s)
Session	Did any code in session match code from another user?	-0.042	1	22	< .0001 ***

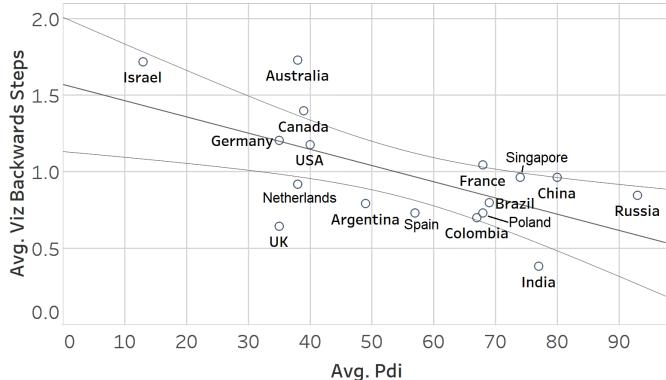


Fig. 3. Average *back-steps* per visualization vs. *PDI*, labeled by country, for countries with at least 10,000 code executions. Linear regression line with confidence bands included.

and low PDI countries. Some observations we made about potential confounds were:

- Each country had multiple sessions with only one code execution and multiple with over 5. We control for this with *# of executions* and *execution number in session*.
- Each country had both short one or two line programs, and longer 20+ line programs. We control for this with *Length of code (# chars)*.
- Each country had programs that were simple and had no functions, and ones that used functions in a complicated way, such as recursion. We control for this with *# steps available* and *# of function calls*.
- Israel, India, and Russia each had one program that implemented a class. We did not control for this because we do not expect it to make a difference.
- Each country had at least one program that matched another program from the dataset. We control for this with *Did any code in session match code from another*

user?.

- The programs across all 80 of our chosen sessions did not match each other, except two for Australia who had very similar text processing code on the same tongue twister. One of these was marked as true for *Did any code in session match code from another user?*.
- Each country had some users making no edits between executions, making small edits between executions, and completely replacing the program between executions. We try to control for this with *Edit-dist. from previous execution*, *Was code just edited?*, and *total edit-distance*.

Given that the programming tasks did not appear to be tied to countries and that we controlled for many of the differences that might exist, we believe our regression analysis to be valid. Therefore our findings suggest that (1) there are significant differences between the use of Python Tutor's back-stepping feature across people from various countries, and (2) a country's Power Distance, which has been previously related to a tendency for self-directed learning, can explain some of these differences.

V. STUDY 2: INDIVIDUAL VALUES AND CODE DEBUGGING

Study 1 showed that users from countries with a low PDI (associated with more self-directed learning) were more likely to back-step in code visualizations than users from countries with a high PDI (H.1). Next we wanted to investigate the role of individually-reported values as opposed to only country-level generalities and do so in a more controlled setting. We therefore launched a second study to investigate how culture relates to back-steps (H.1, H.2), and how back-steps and personal values relate to debugging success (H.3, H.4).

A. Methods

We designed an online experiment as a debugging activity and we embedded the Python Tutor editor and visualizer into our experiment page for participants to use. We advertised our study with a banner on the main Python Tutor website. In our study, all users were given the same content and tasks. We collected demographics and values information from each participant and measured their behaviour in stepping through the debugger and modifying their code. Participants did not receive financial compensation.

1) *Procedure*: Participants provided consent, demographic information, values information (using the 10 question Short Schwartz Value Survey [35]) and then engaged in two time constrained (six-minute) debugging activities by attempting to fix buggy Python code: fixing a broken function to reverse an array, and extracting data from an array of dictionaries. They then were asked follow-up questions about how they used Python Tutor, how they used back-steps, how easy and useful they perceived Python Tutor to be, and how important back-steps were. After completing these questions, participants were shown a score of how many tests their code passed and they could continue working on the problems using Python Tutor if they wished. We included this additional data for testing use of back-steps (H.1 and H.2), but excluded it when testing debugging success (H.3 and H.4).

2) *Analysis*: We conducted mixed-model analyses of variance to test the correlation between back-steps and PDI (H.1) and back-steps and Conservation (H.2). Because 88% of the code execution visualizations had no back-steps and the rest of the data was skewed, we used zero-inflated negative binomial models [36]. Our analysis level was, as in Study 1, on code execution visualizations, with the number of back-steps as the dependent variable. We modeled participantId as a random variable and added either PDI or Conservation as an independent variable (for H.1 and H.2 respectively). We included four more independent variables: number of forward-steps, age,⁵ gender⁶, and reported programming experience (which could influence how they used Python Tutor).

For correlations with debugging success (H.3 and H.4) we conducted mixed-model analyses of variance using Gaussian models. We set the unit of analysis on code execution visualizations with the change in code tests passed for the next visualization as the dependent variable (Δ tests passed)⁷. For independent variables, we used the previous run's passed code tests, the number of forward-steps and the number of back-steps as well as programming experience, which we expected to affect the changes in tests passed. To test for differentiated effects of values aligned with self-learning, we added Conservation along with the interaction between Conservation and back-steps.

⁵Age has previously been shown to influence non-linear navigation [2]

⁶Gender has previously been shown to influence non-linear navigation [2], tinkering [37], [38], and using new features [39]

⁷We also did this with the unit of analysis on problem numbers with the dependent variable as final tests passed, with comparable results.

TABLE II

RESULTS OF *back-step* (H.2) REGRESSION FOR STUDY 2. WE FOUND THAT CONSERVATION WAS MARGINALLY NEGATIVELY CORRELATED WITH THE NUMBER OF BACK-STEPS ($\beta = -0.11$, $p < .089$).

variable	coeff.	z	p
Conservation	-0.11	-1.7	.089 .
# of forward-steps	0.077	12.34	$\leq .0001$ ***
age	-0.011	-1.7	.089 .
gender-Female	-0.36	-1.7	.084 .
gender-Other	0.41	-0.61	.50
Prog. Experience (Linear)	-0.23	-1.2	.23
Prog. Experience (Quadratic)	-0.38	-2.1	.039 *
Prog. Experience (Cubic)	-0.091	0.53	.60
Prog. Experience (4 th)	-0.26	-1.55	.12

3) *Participants*: We ran the study between July and September 2017. During this time, 857 participants completed the demographics and values survey, 522 finished the first problem, and 348 finished the entire activity, providing us with 2,697 visualization sessions. Of those sessions, 2,003 had forward-steps (458 users), and 504 had back-steps (276 users). The average age of users was 27 years ($SD = 12$ years) and 17% identified as female. Users were fairly evenly distributed across five levels of self-reported programming background (Little or none, ≤ 3 months, ≤ 6 months, ≤ 1 year, more). The average Conservation score was -0.71 ($SD = 1.2$) and the country averages ranged from -2.1 to 2.5, representing large differences along the Conservation vs. Openness-to-change dimension. Most participants were from the US (17%), India (17%), China (8.2%), and Russia (4.9%). The average PDI was 61 ($SD = 20.5$), roughly half the highest PDI of 120.

B. Results

For H.2, Conservation was marginally negatively correlated with the number of back-steps in a code execution visualization ($\beta = -0.11$, $p < .089$, see Table II). We also tested the relation between PDI and back-steps (H.1) using a similar model, but with PDI in the place of Conservation. PDI and the number of back-steps were not significantly correlated ($\beta = 0.0034$, $p < .39$).

PDI and Conservation were weakly correlated ($r_{(735)} = .17$, $p < .0001$), though the correlation was much smaller than we expected. This may be due to high variance of participants' values within countries, since when we averaged Conservation values by country, the correlation was higher ($r_{(55)} = .31$, $p < .02$).

While PDI did not explain the differences in the number of back-steps between countries, we did find significant differences between countries, such as between Canada ($M = 1.1$, $SD = 4.0$) and Japan ($M = 0.31$, $SD = 1.1$); $t_{(185)} = 2.02$, $p < .044$.

Our analysis of debugging progress (Table III) showed that back-steps correlate negatively with Δ tests passed ($F_{(1,1692)} = 5.8$, $p = .016$), the opposite of what we predicted in H.3. We additionally found a significant negative coefficient for the interaction of back-step and Conservation

TABLE III
RESULTS OF Δ tests passed (H.3, H.4) REGRESSION FOR STUDY 2.

variable	coeff.	df	F	p
Previous Run Tests Passed	-0.19	1	33	$\leq .0001$ ***
# of forward-steps	0.0059	1	8.5	.0036 **
# of back-steps	-0.023	1	5.8	.016 *
Conservation	-0.014	1	0.30	.59
Programming Experience	N/A	4	13	$\leq .0001$ ***
# of back-steps : Conservation	-0.012	1	4.3	.038 *

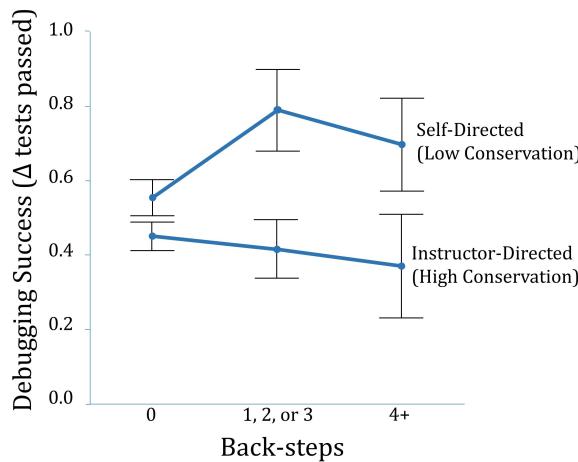


Fig. 4. Average Δ tests passed by number of back-steps and Conservation (mean split) for code executions with at least one forward-step. Bars indicate standard error. Compare to Figure 2 showing H.4.

($F_{(1,1701)} = 4.3$, $p = .038$), confirming H.4 when the negative result of H.3 is accounted for (see Figure 4).

In free-response answers, participants mentioned taking back-steps to check their understanding of code, find the source of bugs, view a set of steps again, and go back to a step they had accidentally skipped over. Some who did not take back-steps mentioned running out of time, finding the problem too easy to require back-steps, or finding forward-steps sufficient.

VI. DISCUSSION

A. Power Distance Negatively Correlates with Back-Steps

Our results demonstrate that the use of non-linear navigation within an online visual debugger varies with national culture: the more a country's people tend to value self-directed learning, the more back-steps they will take. Study 1 confirmed that Power Distance (PDI) negatively correlated with back-steps (H.1), meaning that users in countries with a higher PDI were less self-directed in their use of Python Tutor. This is consistent with the results of a previous MOOC study [2], where users from more student-centered countries were more likely to navigate with non-linear "backjumps". It is also consistent with prior literature on culture and PDI, supporting the claim that PDI is related to self-directed learning [8].

Study 2 revealed no correlation between PDI and back-steps, but showed that Conservation and back-steps are marginally negatively correlated (H.2). This suggests that a

TABLE IV
RESULTS FOR EACH HYPOTHESIS.

	Hypothesis	Outcome
H.1	PDI of a user's country will negatively correlate with the number of back-steps that user takes.	Supported by Study 1. Not supported by Study 2.
H.2	Conservation will negatively correlate with back-steps.	Marginally supported by Study 2.
H.3	Back-steps in code visualizations will correlate with debugging success.	Study 2 found negative correlation instead.
H.4	Conservation will interact with the correlation between back-steps and debugging success.	Supported by Study 2.

user's Conservation score has a larger effect than national culture on the use of back-stepping.

We also found several significant differences between countries in the use of back-stepping. Because the cultural measures PDI and Conservation could not fully explain these differences, it is likely many differences are due to factors other than self-directedness, such national and individual differences in background, experience, socioeconomic status and math competency, and reason for using Python Tutor. PDI and Conservation also may not be sufficient measures of self-directedness to account for the variation we saw.

B. Back-Steps Correlate With Less Debugging Success (Depending on Personal Values)

We were surprised to see that back-steps were negatively correlated with debugging success, the opposite of our hypothesis (H.3). The finding raises doubts about whether back-stepping is helpful in debugging (though there may be other learning benefits to back-stepping that we did not capture). It is possible that instead of measuring back-steps being used in a helpful, intentional way, the back-steps we measured were instead a symptom of struggle [40]. For example, back-steps may have been used in a haphazard way by someone having trouble [41], [42], or as a way of verifying a result they did not believe at first [41].

Finally, we confirmed our last hypothesis (H.4), though we have to modify the phrasing given the result of H.3: For higher Conservation learners, the negative correlation between back-steps and debugging success was stronger, and for lower Conservation learners the negative correlation was weaker. That is, for instructor-directed learners, many back-steps meant less debugging success, while self-directed learners saw less of a relationship between back-steps and debugging success (Figure 4). These results demonstrate that the benefits of some debugging features may vary with personal values for self-directed or instructor-directed learning.

C. Relations to Research on Tinkering and Gender

Our study shares a number of parallels with a previous study on tinkering and gender [37]. Our study supports this prior work in finding a marginally significant trend of females taking

fewer back-steps in Study 2. We then extend that work by showing similar effects to what they found, but with different groups (countries in ours, gender in theirs) varying along different measures of independence (self-directed learning in ours, self-efficacy in theirs) and varying in feature use (back-steps in ours, tinkering in theirs).

The prior work raises further questions about ours, such as: *To what extent are back-steps used in a way that can be considered tinkering? How does membership in different groups interact (e.g., females in India)?* The prior work showed different benefits for exploratory tinkering vs. repeated tinkering, so: *Are there similar different uses of back-steps?*

D. Design Implications

Our results suggest several implications for when back-steps (and other non-linear navigation features) may need to be emphasized, de-emphasized or scaffolded for instructor-directed learners (i.e., those who prefer instructors to direct their learning). Back-steps (and potentially other non-linear navigation) are either detrimental to users or a symptom of struggle. Whatever the cause, this relationship was especially strong for instructor-directed learners. If backward navigation is in fact detrimental to instructor-directed learners, designers may want to de-emphasize or hide backward navigation for those users. Alternatively, if backward navigation is a symptom of struggle for instructor-directed learners, designers may want to provide support and intervention when they detect those learners navigating backwards.

Designers who want to help users across cultures make effective and efficient use of back-steps (or other non-linear navigation features) may need to make these features more prominent. They may also want to provide tutorials or wizards to give instructor-directed users a forward path for learning to navigate backwards (in line with previous suggestions for high PDI countries [21]). Additionally back-steps could be augmented with additional information, such as suggested relevant backward slices of steps for user-selected variables or output (following Whyline [31]), or higher-level holistic views showing context at a glance, providing an alternative way of learning that doesn't involve taking back-steps (following Omnicode [43]).

As evident from the above, programming education tools are unlikely to optimize learning if they are developed in a one-size-fits-all manner. Instead, our results show that people from different countries make different use of key features, suggesting that programming education tools should adapt to preferences and behaviors to optimally support the learner. We showed that PDI and Conservation can be useful as proxy measures for self-directed learning to guide such adaptations, even though they only partially explain the variance between countries. PDI is particularly convenient for designers because it can be derived based on a user's IP address, needing no extra input from learners. Still, designers should be aware that the trends we found for PDI are averaged across large samples, and individual variations may make appropriate adaptation challenging. Prior efforts have worked around this issue by

bootstrapping an initial adaptation with the help of PDI (and other dimensions) before extracting individual information about a user's behavior and preferences from behavioral data [21]. Such an adaptive system circumvents the problem of data sparseness, preventing initial shortcomings for most people, but still updating its priors over time.

VII. LIMITATIONS AND FUTURE WORK

Our study compared use of a single debugger interface feature with one national cultural measure and one individual value measure. Testing only one feature of one code visualization tool limits our ability to generalize to other interfaces. In the future, we plan to further investigate other features of programming education environments, such as information density, cooperative programming, or prominent achievement scores, to evaluate possible effects of country and culture.

In our two studies, we did not directly measure self-directed learning but used proxy measures which may not adequately capture this concept. This is particularly the case with the national measure of PDI, since generalizing by country collapses many meaningful variations between groups of people and individuals. While prior work has repeatedly suggested a link between self-directed learning, PDI, and Conservation, more research is needed to investigate whether these cultural dimensions indeed predict different levels of self-directed learning. This was further complicated by potential bias in our sample from each country. In particular, the subset of visitors to Python Tutor who chose to participate in Study 2 might have different demographics and levels of self-directed learning than those who did not choose to participate.

Future work should also further investigate the benefits, detriments and uses of non-linear navigation, such as back-stepping, especially since our results contradicted our hypothesis that back-steps use would correlate with debugging success. We especially hope to see more work evaluating alternative functionalities that enable users to better learn programming, and evaluate whether such functionalities have a differential effect on debugging success depending on a user's culture.

VIII. CONCLUSION

Our findings show that visual debuggers are used differently by different groups and do not equally benefit all learners. Importantly, we found that these differences can be measured and predicted. We hope that our work will inspire designers and developers to create programming education tools that adapt to their user's cultural backgrounds.

IX. DATA SET

To enable replication and extension of our work, all of the code and data sets from both studies are on GitHub: <https://github.com/kylethayer/culture-debugging-study-data>

ACKNOWLEDGMENT

Special thanks to Jacob O. Wobbrock, Nigini A. Oliveira, Daniel Epstein, Amanda Swearngin, Eunice Jun, William Tressel, Kurtis Heimerl, and Rahul Banerjee.

REFERENCES

- [1] “Code.Org: About Us,” 2018. [Online]. Available: <https://code.org/about>
- [2] P. J. Guo and K. Reinecke, “Demographic differences in how students navigate through MOOCs,” in *Proceedings of the first ACM conference on Learning@ scale conference*. ACM, 2014, pp. 21–30. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2566247>
- [3] J. D. Hansen and J. Reich, “Democratizing education? Examining access and usage patterns in massive open online courses,” *Science*, vol. 350, no. 6265, pp. 1245–1248, Dec. 2015, 00000. [Online]. Available: <http://www.sciencemag.org/content/350/6265/1245>
- [4] C. Sturm, A. Oh, S. Linxen, J. Abdelnour Nocera, S. Dray, and K. Reinecke, “How WEIRD is HCI?: Extending HCI Principles to other Countries and Cultures,” in *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*. ACM, 2015, pp. 2425–2428, 00000. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2702656>
- [5] M. Guzdial, “Limitations of moocs for computing education- addressing our needs: Moocs and technology to advance learning and learning research (ubiquity symposium),” *Ubiquity*, vol. 2014, no. July, pp. 1:1–1:9, Jul. 2014. [Online]. Available: <http://doi.acm.org/10.1145/2591683>
- [6] P. J. Guo, “Non-native english speakers learning computer programming: Barriers, desires, and design opportunities,” in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI ’18, 2018.
- [7] R. F. Kizilcec, A. J. Saltarelli, J. Reich, and G. L. Cohen, “Closing global achievement gaps in MOOCs,” *Science*, vol. 355, no. 6322, pp. 251–252, Jan. 2017. [Online]. Available: <http://science.sciencemag.org/content/355/6322/251>
- [8] G. Hofstede, “Cultural differences in teaching and learning,” *International Journal of Intercultural Relations*, vol. 10, no. 3, pp. 301–320, Jan. 1986. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/0147176786900155>
- [9] R. F. Kizilcec and G. L. Cohen, “Eight-minute self-regulation intervention raises educational attainment at scale in individualist but not collectivist cultures,” *Proceedings of the National Academy of Sciences*, p. 201611898, Apr. 2017. [Online]. Available: <http://www.pnas.org/content/early/2017/04/07/1611898114>
- [10] G. Hofstede, *Culture’s Consequences: International Differences in Work-Related Values*. SAGE, Jan. 1984.
- [11] A. Marcus and E. W. Gould, “Crosscurrents: cultural dimensions and global Web user-interface design,” *interactions*, vol. 7, no. 4, pp. 32–46, 2000. [Online]. Available: <http://dl.acm.org/citation.cfm?id=345238>
- [12] P. J. Guo, “Online Python Tutor: Embeddable web-based program visualization for CS education,” in *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE ’13. New York, NY, USA: ACM, 2013, pp. 579–584. [Online]. Available: <http://doi.acm.org/10.1145/2445196.2445368>
- [13] J. Sorva, *Visual program simulation in introductory programming education*. Aalto University, 2012. [Online]. Available: <https://aaltdoc.aalto.fi:443/handle/123456789/3534>
- [14] S. H. Schwartz, “Universals in the content and structure of values: Theoretical advances and empirical tests in 20 countries,” *Advances in experimental social psychology*, vol. 25, pp. 1–65, 1992. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0065260108602816>
- [15] P. Guo, “Visualize python, java, javascript, typescript, ruby, c, and c,” 2018. [Online]. Available: <http://pythontutor.com/>
- [16] G. Hofstede, “Cultures and organizations: Software of the mind, intercultural co-operation and its implications for survival,” 1997.
- [17] E. Callahan, “Cultural similarities and differences in the design of university web sites,” *Journal of Computer-Mediated Communication*, vol. 11, no. 1, pp. 239–273, 2005.
- [18] B. McSweeney, “Hofstede’s model of national cultural differences and their consequences: A triumph of faith-a failure of analysis,” *Human relations*, vol. 55, no. 1, pp. 89–118, 2002.
- [19] “Pupil-teacher ratio in primary education (headcount basis) | Data,” 2017. [Online]. Available: <https://data.worldbank.org/indicator/SE.PRM.ENRL.TC.ZS>
- [20] “GDP per capita (current US\$) | Data,” 2017. [Online]. Available: <https://data.worldbank.org/indicator/NY.GDP.PCAP.CD>
- [21] K. Reinecke and A. Bernstein, “Knowing what a user likes: A design science approach to interfaces that automatically adapt to culture,” *Mis Quarterly*, vol. 37, no. 2, pp. 427–453, 2013. [Online]. Available: <http://www.misq.org/skin/frontend/default/misq/pdf/V37I2/ReineckeBernstein.pdf>
- [22] A. Bardi and S. H. Schwartz, “Values and Behavior: Strength and Structure of Relations,” *Personality and Social Psychology Bulletin*, vol. 29, no. 10, pp. 1207–1220, Oct. 2003. [Online]. Available: <http://psp.sagepub.com/content/29/10/1207>
- [23] N. T. Feather, “Values, valences, and choice: The influences of values on the perceived attractiveness and choice of alternatives,” *Journal of Personality and Social Psychology*, vol. 68, no. 6, pp. 1135–1151, Jun. 1995. [Online]. Available: <http://offcampus.lib.washington.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=pdh&AN=1995-32996-001&site=ehost-live>
- [24] S. Schwartz, “Value priorities and behavior: Applying a Theory of Integrated Value Systems,” in *The psychology of values: The Ontario symposium*, vol. 8, 2013. [Online]. Available: <https://books.google.com/books?hl=en&lr=&id=DACsdMk7qqaC&oi=fnd&pg=PA1&dq=schwartz+values+behavior&ots=u3nzArGxx6&sig=oe4XQPtH6yS2Rea6kjcYRGLs5h8>
- [25] C. Hofer, M. Denker, and S. Ducasse, “Design and implementation of a backward-in-time debugger,” in *NODE 2006*. GI, 2006, pp. 17–32. [Online]. Available: <https://hal.archives-ouvertes.fr/inria-00555768/>
- [26] B. Lewis, “Debugging Backwards in Time,” *arXiv:cs/0310016*, Oct. 2003, arXiv: cs/0310016. [Online]. Available: <http://arxiv.org/abs/cs/0310016>
- [27] G. Pothier and E. Tanter, “Back to the future: Omniscient debugging,” *IEEE software*, vol. 26, no. 6, 2009. [Online]. Available: <http://ieeexplore.ieee.org/abstract/document/5287015/>
- [28] Z. Azar, “PECCit: An Omniscent Debugger for Web Development,” *Electronic Theses and Dissertations*, Jan. 2016. [Online]. Available: <http://digitalcommons.du.edu/etd/1099>
- [29] S. P. Booth and S. B. Jones, “Walk backwards to happiness: debugging by time travel,” in *Proceedings of the 3rd International Workshop on Automatic Debugging; 1997 (AADEBUG-97)*. Linköping University Electronic Press, 1997, pp. 171–184. [Online]. Available: <http://www.ep.liu.se/ecp/article.asp?issue=001&article=014>
- [30] J. Engblom, “A review of reverse debugging,” in *System, Software, SoC and Silicon Debug Conference (S4D)*, 2012. IEEE, 2012, pp. 1–6. [Online]. Available: <http://ieeexplore.ieee.org/abstract/document/6338149/>
- [31] A. J. Ko and B. A. Myers, “Finding Causes of Program Output with the Java Whyline,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI ’09. New York, NY, USA: ACM, 2009, pp. 1569–1578. [Online]. Available: <http://doi.acm.org/10.1145/1518701.1518942>
- [32] “GeoLite2 Free Downloadable Databases || Maxmind Developer Site,” 2015, 00000. [Online]. Available: <http://dev.maxmind.com/geoip/geoip2/geolite2/>
- [33] “Geert Hofstede | Hofstede Dimension Data Matrix,” 2015, 00002. [Online]. Available: <http://www.geerthofstede.nl/dimension-data-matrix>
- [34] T. Lumley, P. Diehr, S. Emerson, and a. L. Chen, “The Importance of the Normality Assumption in Large Public Health Data Sets,” *Annual Review of Public Health*, vol. 23, no. 1, pp. 151–169, 2002. [Online]. Available: <http://dx.doi.org/10.1146/annurev.publhealth.23.100901.140546>
- [35] M. Lindeman and M. Verkasalo, “Measuring Values With the Short Schwartz’s Value Survey,” *Journal of Personality Assessment*, vol. 85, no. 2, pp. 170–178, Oct. 2005. [Online]. Available: http://dx.doi.org/10.1207/s15327752jpa8502_09
- [36] J. S. Preisser, J. W. Stamm, D. L. Long, and M. E. Kincade, “Review and Recommendations for Zero-inflated Count Regression Modeling of Dental Caries Indices in Epidemiological Studies,” *Caries research*, vol. 46, no. 4, pp. 413–423, 2012. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3424072/>
- [37] L. Beckwith, C. Kissinger, M. Burnett, S. Wiedenbeck, J. Lawrence, A. Blackwell, and C. Cook, “Tinkering and gender in end-user programmers’ debugging,” in *Proceedings of the SIGCHI conference on Human Factors in computing systems*. ACM, 2006, pp. 231–240. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1124808>
- [38] M. G. Jones, L. Brader-Araje, L. W. Carboni, G. Carter, M. J. Rua, E. Banilower, and H. Hatch, “Tool time: Gender and students’ use of tools, control, and authority,” *Journal of Research in Science Teaching: The Official Journal of the National Association for Research in Science Teaching*, vol. 37, no. 8, pp. 760–783, 2000.

- [39] L. Beckwith, M. Burnett, S. Wiedenbeck, C. Cook, S. Sorte, and M. Hastings, "Effectiveness of end-user debugging software features: Are there gender issues?" in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2005, pp. 869–878. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1055094>
- [40] E. S. Tabanao, M. M. T. Rodrigo, and M. C. Jadud, "Predicting At-risk Novice Java Programmers Through the Analysis of Online Protocols," in *Proceedings of the Seventh International Workshop on Computing Education Research*, ser. ICER '11. New York, NY, USA: ACM, 2011, pp. 85–92. [Online]. Available: <http://doi.acm.org/10.1145/2016911.2016930>
- [41] M. C. Jadud, "A First Look at Novice Compilation Behaviour Using BlueJ," *Computer Science Education*, vol. 15, no. 1, pp. 25–40, Mar. 2005. [Online]. Available: <https://doi.org/10.1080/08993400500056530>
- [42] D. N. Perkins, C. Hancock, R. Hobbs, F. Martin, and R. Simmons, "Conditions of Learning in Novice Programmers," *Journal of Educational Computing Research*, vol. 2, no. 1, pp. 37–55, Feb. 1986. [Online]. Available: <http://journals.sagepub.com/doi/10.2190/GUJT-JCBJ-Q6QU-Q9PL>
- [43] H. Kang and P. J. Guo, "Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations," in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. ACM, 2017, pp. 737–745.

Tinkering in the Wild: What Leads to Success for Female End-User Programmers?

Louise Ann Lyon
ETR

Scotts Valley, CA, USA
louann.lyon@etr.org

Chelsea Clayton
ETR

Scotts Valley, CA, USA
chelseab@etr.org

Emily Green
ETR

Scotts Valley, CA, USA
emily.green@etr.org

Abstract— Tinkering has been found to be beneficial to learning, yet women report being disinclined to tinker with software even though their tinkering can be more effective than men’s. This paper reports on a real-world study of how female end-user programmers tinker with new and existing code and what makes their tinkering successful. Findings show that tinkering falls into two main categories: testing an educated guess (more successful) or haphazard trial and error (less successful). In addition, learners occasionally do not tinker to test a successful solution but rather wait to ask another for confirmation of their educated guess before proceeding. Conclusions from this work show that tinkering leads to success when participants are thinking critically about what the code is doing and have hypothesized expected results from code changes. These findings suggest that designers of end-user programmer instructional materials would assist learners by giving explicit tools and techniques that foster successful tinkering.

Keywords— *end-user programming, tinkering, gender*

I. INTRODUCTION

With the rise of the popular Salesforce CRM (customer relationship management) cloud-based software, a new platform for end-user programmers (EUPers) has become widespread. Companies that pay for Salesforce SaaS (Software-as-a-Service) require Salesforce administrator employees (“admins”) to configure the software for their purposes, but when business needs require customization beyond what is available in the point-and-click administrative interface, development work in the Salesforce Apex programming language—similar to Java—is required. To meet these needs, some Salesforce admins are teaching themselves to code in Apex—either to move into development work themselves or to better communicate with and oversee external developer contractors. However, to our knowledge, no research has been done in this real-world setting to investigate the learning activities and strategies of these novice EUPers.

The number of jobs requiring programming skills has increased dramatically in recent times (bls.gov/ooh/), making end-user programming a critical workplace skill. For the many workers who did not study programming during their formal schooling years, the ability to learn this skill while on the job becomes necessary. Finding ways to facilitate this form of informal learning could have far-reaching consequences.

One of the activities of learning to program that some evidence suggests is helpful to novices is tinkering. Research on end-user programmers has found that females’ tinkering can

be more effective than males’ in end-user programmer’s debugging in a lab setting [1]. Researchers in that study noticed that women tended to pause more than men when tinkering, which suggested that the increased effectiveness of the tinkering could be due to participant reflection during these pauses. In work done in a formal learning setting, researchers found that female computer science/computer engineering majors self-reported being disinclined to tinker with software, the authors positing that this was perhaps because the women viewed tinkering as open-ended and without purpose[2].

To fill a gap in this previous work, this paper reports on data from a study that took an ethnographic approach to investigate a virtual, grassroots women’s volunteer coaching and learning group focused on learning Apex. As part of the research, we collected observation and think-aloud data as learners completed homework-like practice problems. We analyzed this data to explore what female EUPers were thinking when they paused while tinkering to find when and how thinking during pauses leads to successful vs. unsuccessful tinkering while working in a real-world setting. The contributions of this work are therefore: (1) a field study to investigate real-world, EUPers who genuinely want to learn to program, (2) a conceptualization of successful and unsuccessful tinkering in this setting, and (3) suggestions for potential strategies that could be given to EUPers to foster successful tinkering. Although results from this work should be applicable to all learners, the focus on a group underrepresented in computing means that findings could be leveraged to broaden participation.

II. BACKGROUND AND RELATED WORK

As part of the research done on End-User Software Engineering (see Ko et al. [5] for a review), one of the aspects of problem solving strategies seen as important in learning to program is “tinkering,” which has been defined in several ways. Berland, Martin, Benton, Smith, and Davis [6], for example, pull together eight definitions—from “playful experimentation” to “just-in-time activity.” Krieger, Allen, and Rawn [2] address tinkering in computer science, creating an initial list of software engineering tinkering that includes exploratory behaviors, deviation from instructions, lack of reliance on formal methods of learning and instruction, and the use of trial and error techniques.

Despite differences in definitions, these works argue for benefits of tinkering and the importance of understanding tinkering behavior to find ways to guide the development of

computer science teaching and learning resources to maximize the beneficial use of tinkering by novice programmers. One of the threads in the tinkering research has been the investigation of gender differences in tinkering behavior and usefulness. Beckwith et al. [1], for example, found that female EUPers tinkered less than males, but that their tinkering was very effective. Krieger et al. [2] furthered this work in a CS education setting, finding that females viewed tinkering as an open-ended pursuit and self-reported that they tinkered less than males. Cao et al. [3] found that code inspection was ineffective for females, but the environment used in their study did not allow comprehensive information processing, unlike the Salesforce environment existing in this study. Burnett et al. [4] found that women were less inclined to tinker among the various populations studied in industry settings, but none of the studies reported on involved novices learning how to code to break into the software development profession.

III. METHODOLOGY

Since we were interested in investigating a real-world phenomenon in context and over which the investigator did not have control, we used a case study methodology in this study. This research follows Yin's [7] definition of "an empirical inquiry that investigates a contemporary phenomenon (the 'case') in depth and within its real-world context, especially when the boundaries between phenomenon and context may not be clearly evident" (pg. 16). The purposes of our research strategy were exploratory and descriptive [8] focused on a naturally-occurring phenomenon to gain insight into what women are thinking as they tinker with new and existing code. Our design was a three-case design with the unit of analysis being a female novice end-user programmer. This design was desirable as the focus was on a depth of understanding of thinking while programming and debugging for women of different backgrounds.

Participants in this study were recruited from the learners in the virtual women's Apex coaching and learning course held during 2017. Information about the research project was given to the 25 learners—all Salesforce administrators—through emailed flyers, postings on the community forum, mentions in the first class sessions by the coaches, and virtual information meetings held by the researcher and the research assistant. First, participants were recruited for interviews. From the nine volunteers to be interviewed, further participation in think-alouds was offered, and five learners ended up scheduling a regular time and participating at this more intensive level.

Over the ten weeks of the Apex coaching and learning sessions, think-aloud data were collected while learners worked on homework-like problems. During one hour each week, participants verbalized their thinking while working in Apex using the GoToMeeting platform while the researcher watched and listened, prompting the learner to continue to verbalize her thinking if she fell silent. Think-aloud data from learners during nine weeks of homework assignments were audio and screen capture recorded, transcribed, and analyzed for tinkering behavior.

The three think-aloud participants that exhibited comparable and consistent episodes of tinkering yet who had

different backgrounds with technology were chosen as the focus of this short paper. None of the learners had coded in Apex before enrolling in the coaching and learning sessions, but one of the participants had coded in Basic, FORTRAN, and Pascal as part of a computer science degree in the 1980s. The two think-aloud participants not included here had consistent issues with completing the homework during the think-aloud sessions and often asked the observing researcher for explicit instruction during the sessions.

Following Sweeney et al.'s [9] qualitative approach to reliability, multiple coders were used to code and analyze the data. Two researchers combed the screen capture and verbal transcripts of think-aloud data looking for instances of tinkering based on a working definition as *a process of trial and error where writing code was interspersed with information foraging* [10]. The primary researcher—who had collected the data—and an additional researcher scoured the data separately, then discussed their interpretations of what constituted instances of tinkering to consider various standpoints and perspectives. After example instances were agreed upon, one researcher examined the remaining screen capture and transcript data and coded instances of tinkering using the agreed-upon examples as a guide. We created and refined a code hierarchy as part of our data analysis, as is common in qualitative work, iterating both top-down and bottom-up in which more detailed codes were combined under a larger, umbrella code (or theme). We found that all codes for attempted tinkering fell under two large categories: successful tinkering when participants made the desired changes and unsuccessful tinkering when they were unable to make desired changes during the think-aloud period. These two became top-level themes. Under those were such codes as "developer guide example copied/modified," "inspect and change previous line of code," "read and consider error message," etc. In addition to this, we had a low-level code of "check before trying," which became out "missed opportunities to tinker." The analysis method allowed for building consensus through understanding and exploring different perspectives on the data, as contrasted to inter-rater reliability, which is appropriate in more limited circumstances.[9] This data analysis technique provided triangulation of the data and allowed for a rich description and deep understanding of the data. The final write-up of the cases and the illustrating tinkering instances were member checked by participants for correctness.

IV. FINDINGS

Victoria (all names used here are participant-chosen pseudonyms) worked as a Sales Operations Manager at a service security start-up at the time of this study. She was a 32-year-old self-identified Asian American who held a bachelor's degree in economics and communication. Victoria was interested in learning Apex after turning down too many of her boss' requests for Salesforce functionality; she reports that she got kind of frustrated with having to always say "no, no, no we can't do it because..." Once you know how to do Apex, then the answer is always "yes" and you just go out and build it yourself. In addition, Victoria wanted to understand enough Apex to be able to oversee external developer consultants and possibly to modify code that they had written. She did not

anticipate a job change or promotion with a knowledge of Apex; she simply wished to add to her skill set.

At the time of this study, Kate was 54 years old; she self-identified as Caucasian or white. As a young woman, Kate had earned a bachelor's degree in CS and a master's degree in environmental science, and had been a programmer in the 80's. At the time of joining the women's coaching and learning group, Kate had resigned from her admin job to focus on becoming a Salesforce developer because *developers are a great career move*, and her participation was a first step on her path to certification and a job as a developer.

Lila, born in the U.S. of Indian-born parents, had a bachelor's degree in international affairs and a master's degree in healthcare administration. At the time of this study, she and a partner had started their own company offering freelance Salesforce services. In an interview, Lila expressed interest in learning Apex because *it would be an amazing skill* and learning it was something she was *really, really interested in*.

A. Successful: Testing an Educated Guess

Analysis across participants shows that they successfully learned how to write working code when their tinkering was based in an educated guess and trying out code to see what worked or did not work to learn what was possible. Some examples that led to this conclusion follow.

a) *Victoria: week 4:* On occasion, Victoria tried out a piece of code that she was not sure would work before she looked up whether or not the code should run correctly. When this tinkering was done on a targeted piece of code that ran correctly before the change, Victoria learned what worked and she did not struggle with debugging. In week 4, for example, Victoria wrote a line of code that printed a message. In the call to the print method, Victoria appended the name of a Map variable, saying *I actually wonder what will happen* as she types in the variable name. When she runs the code, she sees that it prints out the key and values for each entry in the Map.

b) *Lila: week 4:* In an episode during week 4, Lila tinkered to test the case sensitivity of variable names in Apex. *One thing I've been wondering with myself is, what do they call it, Camel Casing?...if I made it lowercase, is that a typo? I am just going to see what happens.* At this point, she changes the code so that the case does not match. *So, there's no problem that comes up when I saved it, but I wonder if it will execute? Just cause it really has been bugging me lately. I can answer my own question just by testing it.* She then executes the code successfully. *Wow. So, maybe it's just the Camel Casing, must just be for ourselves because it looks like it runs fine. So, really, it's just a visual thing.*

c) *Kate: week 5:* During Kate's week five homework session, she encountered a problem printing out an account name that she had not previously extracted from the database. As she typed her code she mentioned that she may not be able to include the name because she didn't select the account name in a previous line of code. She ran the debug and, indeed, the account name did not print. She returned to her assignment, added the AccountId syntax to the code and her

code then ran successfully. Kate explained, *I was thinking that the Account ID would just be there because it's an ID, but it's not the AccountId that's there. It's the OpportunityId that's there, and so to get the AccountId, I actually have to select it.*

B. Unsuccessful: Haphazard Trial and Error

Participants were unsuccessful in writing, running, and learning to code when their tinkering was haphazard, and they changed or added code without thinking through a reason for the changes and what results would be expected, as described in the examples below.

1) *Mistaken use of example code:* Example code was commonly referred to and even copied and pasted as the women worked on their practice problems. However, when the code used was not appropriate for solving the problem, participants could get mired in newly-introduced problems.

a) *Victoria: week 5:* Victoria often looked for examples to model as she worked on her practice code, referring to the Salesforce developer guide, blogs, and her own notes to find code that was similar to what she was attempting to write. However, when she modeled what she was doing after the example code, she sometimes focused on sections of the example that were irrelevant to the task she was attempting to perform, which resulted not only in not giving her the functionality she desired, but also created errors that she did not understand. In week 5, for example, Victoria focused on the return value of an example method, which was unrelated to the bug in her code, and so she modified her method to return a list of accounts instead of a void method. Not only did this not fix the problem, but it also caused an error message about the method signature that she did not understand, causing her to spend additional time debugging to fix the bug she introduced.

2) *Haphazard deleting or adding code:* Occasionally, participants appeared to have no plan of how to proceed writing new code or debugging existing code, and out of apparent desperation they added and deleted symbols or code snippets to try and figure out how to proceed.

a) *Lila: week 5:* Lila was working on a homework assignment that was given with a bug in the code. As Lila struggled to debug the file, she tinkered with the code without any clear plan, such as changing a plus sign to a comma, adding a system debug line of code, and deleting a plus sign. She recognized that this tinkering may have caused her more problems than it solved, saying *I've deleted a few things and a few times that I'm hoping haven't totally messed it up. That I'm actually just getting an error in addition to the fact that, one, it's probably already broken, and then, secondly, that I have also done something else wrong.* With no success at debugging after 45 minutes, Lila said she *feels slightly depressed*, and moved on to trying to solve another problem.

C. Missed Opportunities to Tinker

Data for this study include episodes in which the participants had an opportunity to tinker to further their learning and/or to

try out a hunch. As the instances here show, however, on some occasions the women wrote lines of code in a format familiar to them instead of tinkering with a new idea or they made a note to ask a coach for an answer rather than trying something that they thought might work.

a) *Victoria: week 2:* During the second week of the course, one of the practice exercises asked learners to add two numbers to a list of integers that had been created in existing code. Victoria first tried calling the List method “.add()” function with two numbers separated by commas; when this did not work, Victoria correctly separated the code into two function calls, each with one number in parentheses. However, she could not see that her code worked, because the list in the output showed the existing list elements followed by an ellipsis to indicate that not all elements were showing. Although Victoria suspected that this might be so (*I'm not sure why it's not showing up and at first I was thinking, oh, the dot dot dot, maybe because it didn't fit, so maybe it's in there.*), she made a note that she wanted to ask why her code did not work instead of tinkering with the code to test her educated (and correct!) guess.

2) *Lila: week 2:* When attempting to insert numbers into a List, Lila first typed in “numberList.add = 45;” then inserted parentheses (“numberList.add() = 45;”), after which she saw the compiler error message saying “Method does not exist or incorrect signature: [List].add().” Lila went back to the code and properly corrected it, but expressed a disinclination to tinker, saying *now I just need to confirm it. I guess I could test it, but I'm more of somebody who wants to confirm that I have it right.*

3) *Victoria: week 4:* In week 4, Victoria said *I'm more familiar with this so I'm just going to stick to what I know* as she proceeds to declare a List variable, then to write two .add() statements to add strings to the list instead of declaring and defining the variable in the same line of code.

V. DISCUSSION

This study is a first to examine how female novice EUPers tinker with new and existing code in the real world, and what makes their tinkering successful. For women learning to code to increase workplace skills, their tinkering behavior—although sometimes curiosity-based [1]—is not as much playful as it is trial and error with just-in-time research [10]. For women in this study, tinkering was most successful when learners were examining the code, critically thinking about what each piece of the code did, making an educated guess about what might change the code in the manner they would expect, and then making a targeted change to test their hypothesis. This behavior was exhibited most consistently by Kate, who had a programming background and who could confidently make changes without being flustered. Tinkering was least successful when learners haphazardly deleted or added symbols or copied and pasted examples that implement unrelated functionality. Both Victoria and Lila exhibited this behavior, making changes without understanding all the details of the code.

Our findings support work that notes a tendency by females to ask for help more often than males while learning to program [2], showing that in some cases participants ask others rather than tinkering to test a hunch. It also supports work that suggests that women are interested in tinkering that facilitates understanding of how to program [2]. Our findings add context and detail to work that found a tie between female tinkering and understanding [1], showing when pauses during tinkering lead to success. This work also extends work on novice programmer example use [11] to an adult, work-focused population, finding that ‘example comprehension hurdles’ exist in this population, as well, when such participants are not able to locate and understand the critical components of an example.

Limitations of this study are consistent with other work of this type, including possible participant behavior change based on the presence of an observer and the call for caution in generalizing this work to other female EUPers learning while in the workforce.

Based on findings from this work, creators of content and materials used by EUPers learning for workforce development reasons may want to make recommendations to users for techniques that are successful when tinkering or that guide users to useful tinkering techniques. Strategies that were helpful to these women would be helpful to other EUPers and could be explicitly explained or demonstrated in learning materials. One such strategy is to always have running code so that any changes made while tinkering are the only sources of possible problems, forcing focused debugging. Another strategy useful on its own or with the previous one is commenting out sections of code rather than deleting as code is tested; a strategy that Lila did not follow, causing her to spend 45 minutes of unsuccessful tinkering. EUPer materials could explain that each word and symbol is significant, and that programmers should carefully examine what each contributes to a line of code, planning out what needs to be done, and trying out small changes, working from what is known to what needs to be tried and adding first within comment symbols that can later be removed. How to use resource materials could also be explained in learning materials, such as noting what search terms are likely to be most useful when searching the developer guide.

By explicitly giving tips and tricks for tinkering, perhaps women might come to view it as tangibly helpful to their learning, and therefore might be more inclined to tinker, which in turn might help their software development learning. It should not be too late for women in the workplace to discover an interest in programming, thereby making a career change helping to diversify the computing workforce.

ACKNOWLEDGMENT

Thank you to the steering committee, coaches, and learners of the Apex women’s coaching and learning group for their enthusiastic participation in this study. This work was supported by NSF under grant #1612527. Any opinions, findings, conclusions, or recommendations are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES

- [1] L. Beckwith *et al.*, "Tinkering and gender in end-user programmers' debugging," in *Proceedings of the SIGCHI conference on Human Factors in computing systems*, 2006, pp. 231-240: ACM.
- [2] S. Krieger, M. Allen, and C. Rawn, "Are females disinclined to tinker in computer science?", in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, 2015, pp. 102-107: ACM.
- [3] J. Cao, K. Rector, T. H. Park, S. D. Fleming, M. Burnett, and S. Wiedenbeck, "A debugging perspective on end-user mashup programming," in *2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2010, pp. 149-156: IEEE.
- [4] M. Burnett *et al.*, "Gender differences and programming environments: across programming populations," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2010, p. 28: ACM.
- [5] A. J. Ko *et al.*, "The state of the art in end-user software engineering," *ACM Computing Surveys (CSUR)*, vol. 43, no. 3, p. 21, 2011.
- [6] M. Berland, T. Martin, T. Benton, C. Petrick Smith, and D. Davis, "Using learning analytics to understand the learning pathways of novice programmers," *Journal of the Learning Sciences*, vol. 22, no. 4, pp. 564-599, 2013.
- [7] R. K. Yin, *Case study research: Design and methods*. Sage publications, 2013.
- [8] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.
- [9] A. Sweeney, K. E. Greenwood, S. Williams, T. Wykes, and D. S. Rose, "Hearing the voices of service user researchers in collaborative qualitative data analysis: the case for multiple coding," *Health Expectations*, vol. 16, no. 4, 2013.
- [10] B. Dorn and M. Guzdial, "Learning on the job: characterizing the programming knowledge and learning strategies of web designers," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2010, pp. 703-712: ACM.
- [11] M. Ichinco and C. Kelleher, "Exploring novice programmer example use," in *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*, 2015, pp. 63-71: IEEE.

Exploring the Relationship Between Programming Difficulty and Web Accesses

Duri Long
 Georgia Institute of Technology
 Atlanta, GA, USA
 duri@gatech.edu

Kun Wang
 UNC-Chapel Hill
 Chapel Hill, NC, USA
 wangkl@cs.unc.edu

Jason Carter
 Cisco Systems-RTP
 Raleigh, NC, USA
 jasoncartercs@gmail.com

Prasun Dewan
 UNC-Chapel Hill
 Chapel Hill, NC, USA
 dewan@cs.unc.edu

Abstract—This work addresses difficulty in web-supported programming. We conducted a lab study in which participants completed a programming task involving the use of the Java Swing/AWT API. We found that information about participant web accesses offered additional insight into the types of difficulties faced and how they could be detected. Difficulties that were not completely solved through web searches involved finding information on AWT/Swing tutorials, 2-D Graphics, Components, and Events, with 2-D Graphics causing the most problems. An existing algorithm to predict difficulty that mined various aspects of programming-environment actions detected more difficulties when it used an additional feature derived from the times when web pages were visited. This result is consistent with our observation that during certain difficulties, subjects had little interaction with the programming environment, they made more web visits during difficulty periods, and the new feature added information not available from features of the modified existing algorithm. The vast majority of difficulties, however, involved no web interaction and the new feature resulted in higher number of false positives, which is consistent with the high variance in web accesses during both non-difficulty and difficulty periods.

Keywords— *interactive programming environments, affective computing, distributed help, web foraging, intelligent tutoring*

I. INTRODUCTION

Automatic detection of programming difficulty could be used for several purposes including offering help to students and co-workers in academic and industrial contexts [1], identifying bugs [2, 3], determining the difficulty of tasks, programming constructs, and APIs [2, 4], and determining when automatic hints should be displayed by intelligent tutoring systems [5]. This field is part of the larger area addressing detection of task difficulty. A related subfield is detection of difficulty in web searches [6]. These two subfields have been investigated separately even though they are related in that web searches can be subtasks of programming tasks.

This work brings together these two fields by providing a preliminary answer to the following general question: What is the relationship between difficulty and web accesses in programming tasks that can benefit from web searches? We answer this general question by addressing the following related

sub-questions regarding such tasks: To what extent do developers use web searches to solve problems and how successful are these searches in solving their problems? What kinds of problems are not resolved completely by such searches? How, to what extent, and why can information about web searches be used to improve an existing state of the art online algorithm for predicting and resolving programming difficulties? What kind of data can help answer these questions?

The remaining sections address these questions based on related work and data gathered from a study.

II. RELATED WORK AND BASELINE

A variety of features have been shown to correlate with programming difficulty and the related emotions of frustration and happiness in programming-based tasks. These features have been derived from a variety of information sources including the programming language constructs used [4], interaction with the programming environment [7], output of a Kinect camera [1], and output of eye-tracking electrodermal activity and electroencephalography sensors [3]. Some correlation algorithms have targeted code-comprehension [3], while others have considered code-creation [7]; and some have focused only on correlation [4], while others have also built models to predict difficulties of programmers [3, 7]. Some of predictive systems make inferences incrementally [7], during the task, while others do so after the completion of the task [3].

Features that correlate with and can be used to predict difficulty in web searches have, similarly, been drawn from a variety of information sources including users' search queries, clicks on search results, bookmarked web pages, mouse movements and scroll events, time spent on a web page, and task completion time [6]. Web searches can be performed as stand-alone tasks (such as reading information regarding a medical condition) or they can be sub-tasks of a larger task such as programming.

Web-supported programming has been studied in previous work. Fishtail is a system that recommends arbitrary web pages on the internet based on programming constructs on which the developer is working, but it does not have high accuracy [8]. Reverb is a similar system that achieves higher accuracy by restricting itself to web pages visited earlier by the developer [9]. To illustrate, in Reverb each method call in the current context is mapped to a set of keywords uniquely describing the call, and this set is used to match web pages.

The recommendations provided by Fishtail and Reverb, as well as search engines, consider only how relevant certain components of web pages are to the task at hand, which can be a function of several parameters including how well these components match a manually or automatically generated query, when the page was last updated, and, in case of Reverb, also when the developer last visited the page. Jin, Niu and Wagner [10] consider also the time cost of finding the relevant information in the returned page, which they argue is a function of two important features – the time taken on average to read the entire page and the shape of the expected foraging curve of the page, which plots information gain as a function of the time spent on the page. Each recommendation displayed to the user contains these two features to allow more efficient recommendations to be chosen. For an unknown page, a default foraging curve is displayed. For a known page, the foraging curve depends on whether it contains ranked answers (e.g. StackOverflow), a list of items (e.g. API documentation), a wiki/blog explaining a concept, or a forum with unranked answers. Users are expected to choose, for instance, a ranked answer site over a wiki.

The works above aim to reduce the time required to consult the web to solve software engineering problems. At least one study has found that some such problems – in particular, in web design – are not solved by web searches [11]. To the best of our knowledge, no previous work has used web accesses to predict difficulties with web-supported programming. Thus, our work addresses a new dimension in such programming.

As mentioned earlier, features other than web accesses have been considered by previous works for predicting programming difficulties. The only research that has addressed incremental prediction of difficulty in programming tasks involving code creation consists of several related algorithms, developed by our team, which mine logs of interaction with the programming environment and/or videos captured by a Kinect camera [1]. Only one of these algorithms [12] has an online implementation – that is, an implementation that processes live data, and thus can (and has) actually be used to detect, communicate, and ameliorate programming difficulty incrementally. Other algorithms have offline implementations validated using logged data. Because of space limitations, we use the online algorithm as the only baseline for determining the effectiveness of mining web accesses. We will refer to our implementation of this algorithm as the baseline system.

The baseline system is targeted at the Eclipse IDE and extends the Fluorite tool [13] to capture Eclipse commands. It divides the raw interaction into segments and calculates, for different segments, ratios of the following classes of commands: edit, debug, focus (in and out of the programming environment), and navigation within the programming environment. The training data for the algorithm is passed through the Weka SMOTE filter [14] to artificially increase the members of the minority class (difficulty). It uses the Weka J48 decision tree implementation to make raw inferences. Our algorithm operates on the assumption that the perception of difficulty does not change instantaneously. Therefore, it aggregates the raw predictions for adjacent segments to create the final prediction, reporting the dominant status in the aggregated segments. In addition, it

makes no predictions from the first few events to ignore the extra compilation and focus events in the startup phase.

III. STUDY

An important case of web-supported programming is a task that involves the use of one or more APIs that are well documented on the web. Our study involved such a task – use of the Java AWT/Swing API to create an interactive graphical user interface. The main subtask was to create a program that composes a single or double-decker bus from rectangles and circles. To support input, the subjects were asked to allow users to provide keyboard input for moving the bus, and mouse input for converting a single-decker bus to a double-decker. An additional subtask, designed to pose algorithmic challenges, required subjects to draw a transparent square with yellow borders around the bus and ensure that the bus could not be moved outside the square.

Fifteen graduate and advanced undergraduate students at our university, many of whom had previously held industry internships, participated in the study. Each participant was given at least an hour and a half to complete as many subtasks as possible using Eclipse. On average, they spent about two hours working on the task. We have not yet determined to what extent and in what order they completed the subtasks. They were free to use the Internet to solve their problems. Our baseline system was used to capture interaction with Eclipse. A Firefox plug-in was used to determine web access information. As shown below, each web access contained three pieces of information – its time, the URL visited, and the input search string or exact URL that triggered the visit.

9/13/2013 16:20:47 PM

swing - Java keyListener - Stack Overflow|
<http://stackoverflow.com/questions/11944202/java-keylistener>

During the task, the base system showed the programmers the predictions that it made. The difficulty notification was called “slow progress” and indicated that the programmers were making slower than normal or expected progress. Subjects were provided with a user interface through which they could correct a prediction and/or ask for help. When participants asked for help, they were instructed to indicate what they had done to solve the problem so far and discuss their issue with the third author or another helper. Help was given in the form of URLs to documentation or code examples. Given enough time, many difficulties can be eventually solved. We considered a difficulty insurmountable if the programmers did not think they could solve the problem within the given time constraints. In this study, help requests were considered insurmountable difficulties – other difficulties were considered surmountable. We used information about the predictions, corrections, and help requests to determine ground truth.

IV. WEB ACCESSES AND DIFFICULTY

To what extent did developers use web searches to solve their problems and how successful were these searches? To answer this question, we divided the web accesses of the subjects into web episodes, which are a series of web accesses with no intervening interaction with the programming environment. Let us assume that each web episode was used to address a set of related problems faced by the subject when the

episode started, and at the end of the episode the developer either felt the problem was solved, or they continued to face a surmountable or insurmountable difficulty. Let us also assume, conservatively, that if a difficulty segment had one or more web episodes, then the last web episode was not effective. This is a conservative estimate as the difficulty may not have resulted in a web episode. We found that the percentage of episodes that were ineffective was 8 percent, and the fraction of web episodes that were ineffective and led to insurmountable difficulties was 5 percent. Thus, while the vast majority of web episodes were successful, some did not solve the problem, and some even resulted in help requests, which motivates the design of both better web foraging tools and better tools for detecting difficulty.

What were the topics of unsuccessful searches? We found that the majority of sites visited during difficulties fell into one of three distinct categories: 1) API-related sites (primarily consisting of Java AWT/Swing introductory tutorials, sites related to how to represent 2D-graphics using Java AWT/Swing, sites discussing how to listen to keyboard, mouse, and button events, and sites related to non-graphical components of AWT/Swing such as JPanel), 2) design related sites (primarily Model-View-Controller tutorials), and 3) non-Swing/AWT Java related searches (mostly tutorials).

Table I shows the percentage of different types of topics searched during periods of difficulty, thereby characterizing the difficulty of these topics and/or the adequacy of the documentation about them. More importantly, it helps us understand the nature of some of the search-based difficulties faced by the programmers. It is consistent with user comments in [11] that some (uncharacterized) searches do not solve web-design problems. The fact that 2-D Graphics causes the most difficulty is likely because it involves overriding the paint() method and explicitly calling the repaint() method from a thread that is different from the one that executes the paint() method. We have not yet analyzed the videos to determine the exact causes of the difficulties.

TABLE I. CHARACTERIZING UNSUCCESSFUL WEB SEARCHES

Topic	Searched During Difficulties
API (overall)	75%
Tutorials	17%
2D-Graphics	38%
Events	10%
Components	10%
Design	4%
Java	2%
Other	19%

V. PREDICTIONS USING WEB ACCESSES

How, to what extent, and why could information about web searches improve our baseline online algorithm for predicting these difficulties? To answer this question, we extended the baseline algorithm with an additional feature representing the number of web links traversed during a segment - *weblinktimes*.

We performed the following aggregate analysis to compare the two algorithms. The sizes of the warm-up phase and segments were fixed at 50 and 100 events, respectively, for both algorithms. Moreover, 5 segments were aggregated in both cases. We used 10-fold cross-validation on combined logs of all participants. Only 6% of the statuses were difficulties – that is, facing difficulty was a rare event, which is to be expected if programmers are given tasks they are qualified to tackle with the available resources. The SMOTE filter was used to equalize the number of difficulty and non-difficulty segments in the training data.

10-fold cross-validation assumes all partitions are independent, which may not be the case in our study because the programmers can be expected to learn as they progress through the solution as they better understand the information gathered from the web visits. On the other hand, it is not clear this learning effect changes the pattern of interaction around difficulties – our online algorithm has been used in multiple lab studies and a field study [7] to successfully find difficulties. More importantly, different search strings and web episodes likely had some independence, and as Table I shows, the topics searched had a large degree of independence. Finally, 10-fold cross validation is more comprehensive than 1-fold validation and any dependence between the partitions should make our results conservative rather than inflated.

The true positive and negative rates of the baseline algorithm were 20% and 100%, respectively, that is, the baseline algorithm correctly predicted 20% of the difficulty statuses, and all of the non-difficulty statuses. The web-based algorithm increased the true positive rate substantially to 93%, but reduced the true negative rate slightly to 95%. Based on these two measures, the results are impressive. A more critical analysis is provided by the precision and recall measures. Recall is the same as the true positive rate. Precision is the fraction of flagged positives that were actual positives. The precision of the baseline and modified algorithms are 100% and 54%, respectively. This result is consistent with the inverse relationship seen between precision and recall in other work. The F-score has been devised to give equal weight to both and is defined as:
$$\frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$
. The F-scores of the baseline and modified algorithms are 33% and 71%, respectively. Intuitively, the better F-Score of the modified algorithm can be explained by considering detection of difficulties as being analogous to finding needles in a haystack. In our modified algorithm, one has to search not the whole haystack, but a subset of items whose size is about twice the number of actual needles, which arguably, is a very good result. Of course, how the two measures should be weighed depends on the cost of manually separating false and true positives, which in turn, depends on several factors including whether the helpers are face-to-face with the worker and how much context they have if they are distributed [15].

Why did the new feature make such a difference? We extended an interactive visualization tool we developed earlier [16] to help us understand the impact of weblinktimes in specific difficulties. The data of a participant is represented using our tool in Figure 1. The segment numbers, identified by their start times, are shown on the X-axis. The Y-axis shows different kinds of information about these segments. The four ratios used

in our algorithms as features (as well as some additional ratios) are plotted on the Y-axis using different colors. The code W(number) shows the weblinktimes of the corresponding segment. The row of Predicted bars indicates the status inferred by the online mechanism and the row of Actual bars represents the ground truth. The green bars represent normal progress and the pink bars represent difficulty points. Currently we do not show the status inferred by offline algorithms.



Figure 1. All ratios 0% at beginning of session

This figure shows that during a difficulty missed by the baseline system at the beginning of the interaction of a user, all ratios were at 0%, and the user visited eight web links, perhaps because the user did not know how to start the project. The number of visits is higher than normal - the average and standard deviation of weblinktimes in non-difficulty (difficulty) segments was 0.6 (2.4) and 2.8 (5.3), respectively. These data and this example show that high weblinktimes can be an indication of difficulty, and that this feature is particularly important for detecting difficulties when there is little or no interaction with the programming environment

Can difficulties be associated with low or even zero weblinktimes? 17% of the difficulty segments had no web access. The associated difficulties were presumably caused by algorithm rather than API related issues. Conversely, 15% of the non-difficulty segments had web accesses, which probably prevented burgeoning problems from blossoming into expressed surmountable or insurmountable difficulties. Thus, difficulties may be associated with higher or lower than normal weblinktimes. Intuitively, going to be web may imply either that the developers are lost, or are purposeful, knowing what they are looking for to solve potential problems.

The fact that the vast majority of difficulty and non-difficulty segments had no web access show that weblinktimes, alone, is not sufficient to predict difficulties reliably. As we see above, even when weblinktimes was combined with the features of the baseline system, the precision was low. This is consistent with the average and standard deviations of weblinktimes in non-difficulty and difficulty segments - there is a high overlap in the range of weblinktimes in these two kinds of segments.

We included the focus feature in our base algorithm to account for web searches. So why did adding weblinktimes change our results? There are several reasons. Different focus events can result in different number of web pages being visited. Moreover, web searches may not be preceded by a focus (out) event (Figure 1). Finally, such an event may not result in interaction with the browser, and even if it does, the resulting

web episode may have a varying number of web page visits, as we have seen. In our study, 6% of the segments with non-zero weblinktimes had a focus ratio of zero, and conversely, 83% of segments with non-zero focus ratios had zero weblinktimes. In the remaining segments, we calculated the division of weblinktimes by focus ratio, which had an average of 0.78 and standard deviation of 1.05. Thus, both in theory and practice, focus ratio and weblinktimes do not have an obvious correlation.

VI. DISCUSSION

This study is too specific and small to make general and final conclusions regarding the relationship between web accesses and programming difficulties. Its main contribution is providing first insights on this topic, which, arguably, have implications beyond the specific experiment.

Some of these are independent of difficulty prediction and are tied to the use of the general programming abstractions of widget composition, graphical output, and event-based input – supported in all (UI) toolkits known to us, and of particular relevance to this conference. These contributions include the percentage of difficulties involving web accesses, the relative frequency of web searches of toolkit topics during difficulty periods, and the number of web accesses during difficulty and normal periods. Unlike the numbers for these metrics, the concepts of these metrics are study-independent.

Our prediction-related contributions similarly have experiment-dependent and independent aspects. The general contributions include the idea of using weblinktimes as a new prediction feature, using well-known machine-learning metrics to determine the aggregate impact of adding this feature, exploring the relationship between it and the related existing prediction feature of focus ratio, and extending and using a special visualization tool to understand the impact of adding this feature. The actual prediction-related numbers we present, of course, can be expected to vary in different experiments.

It would be useful to do further analysis with our data such as using leave-one-out analysis rather than cross-validation, which will not have the issue of learning effect. Additional experiments can involve other forms of web-supported programming such as distributed programming. It would be useful to evaluate the prediction value of other features about web accesses such as time spent on a web page [6], the shape of the foraging curve associated with the page [10], and the topic of the page (e.g. 2D-graphic events). Future work can also use these features to make additional kinds of predictions such as whether a difficulty is surmountable or not. An in-depth manual analysis of the videos around difficulty points can reveal additional insights into the difficulties and how to predict them. Finally, it would be useful to implement an online algorithm based on web-accesses that is part of a workflow that also includes implementations of the related work on web-link recommendation [8, 9] and classification of web pages based on foraging cost [10].

This work provides a basis to explore these novel research directions.

ACKNOWLEDGMENTS

Reviewer comments had a major impact on the final paper.

REFERENCES

- [1] Carter, J., M.C. Pichiliani, and P. Dewan, Exploring the Impact of Video on Inferred Difficulty Awareness, in Proc. 16th European Conference on Computer-Supported Cooperative Work. 2018, Reports of the European Society for Socially Embedded Technologies.
- [2] Dewan, P. Towards Emotion-Based Collaborative Software Engineering. in Proc. CHASE@ICSE. 2015. Florence: IEEE.
- [3] Fritz, T., A. Begel, S. Mueller, S. Yigit-Elliott, and M. Zueger. Using Psycho-Physiological Measures to Assess Task Difficulty in Software Development. in Proceedings of the International Conference on Software Engineering. 2014.
- [4] Drosos, I., P. Guo, and C. Parnin. HappyFace: Identifying and Predicting Frustrating Obstacles for Learning Programming at Scale. in IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). 2017. Raleigh, NC: IEEE.
- [5] Price, T.W., Y. Dong, and D. Lipovac. iSnap: Towards Intelligent Tutoring in Novice Programming Environments. . in ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17). 2017. ACM.
- [6] Arguello, J. Predicting Search Task Difficulty. in Proceedings of the 36th European Conference on IR Research. 2014. Springer International Publishing.
- [7] Carter, J. and P. Dewan. Mining Programming Activity to Promote Help. in Proc. ECSCW. 2015. Oslo: Springer.
- [8] Sawadsky, N. and G. C. Murphy, " Fishtail: From task context to source code examples. in Proc. of the 1st Workshop on Developing Tools as Plug-ins. 2011. ACM.
- [9] Sawadsky, N. and G.C. Murphy. Rahul Jiresal: Reverb: recommending code-related web pages. in Proc. ICSE. 2013.
- [10] Jin, X., N. Niu, and M. Wagner:. Facilitating end-user developers by estimating time cost of foraging a webpage. : 31-35. in Proc. VL/HCC. 2017. Raleigh: IEEE.
- [11] Dorn, B. and M. Guzdial. Learning on the Job: Characterizing the Programming Knowledge and Learning Strategies of Web Designers. in Proc. CHI. 2010. ACM.
- [12] Carter, J. and P. Dewan. Design, Implementation, and Evaluation of an Approach for Determining When Programmers are Having Difficulty. in Proc. Group 2010. 2010. ACM.
- [13] Yoon, Y. and B.A. Myers. Capturing and analyzing low-level events from the code editor. in Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools. 2011. New York.
- [14] Witten, I.H. and E. Frank, Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations. 1999: Morgan Kaufmann.
- [15] Carter, J. and P. Dewan, Contextualizing Inferred Programming Difficulties, in *Proceedings of SEmotion@ICSE, Gothenburg*. 2018, IEEE.
- [16] Long, D., N. Dillon, K. Wang, J. Carter, and P. Dewan. Interactive Control and Visualization of Difficulty Inferences from User-Interface Commands. in *IUI Companion Proceedings*. 2015. Atlanta: ACM.
-

A Large-Scale Empirical Study on Android Runtime-Permission Rationale Messages

Xueqing Liu, Yue Leng

*Department of Computer Science
University of Illinois, Urbana-Champaign
Urbana, IL, USA
xliu93,yueleng2}@illinois.edu*

Wei Yang

*Department of Computer Science
University of Texas, Dallas
Richardson, TX, USA
weiyang.utd@gmail.com*

Wenyu Wang, Chengxiang Zhai, Tao Xie
*Department of Computer Science
University of Illinois, Urbana-Champaign
Urbana, IL, USA
{wenyu2,czhai,taoxie}@illinois.edu*

Abstract—After Android 6.0 introduces the runtime-permission system, many apps provide runtime-permission-group rationales for the users to better understand the permissions requested by the apps. To understand the patterns of rationales and to what extent the rationales can improve the users’ understanding of the purposes of requesting permission groups, we conduct a large-scale measurement study on five aspects of runtime rationales. We have five main findings: (1) less than 25% apps under study provide rationales; (2) for permission-group purposes that are difficult to understand, the proportions of apps that provide rationales are even lower; (3) the purposes stated in a significant proportion of rationales are incorrect; (4) a large proportion of customized rationales do not provide more information than the default permission-requesting message of Android; (5) apps that provide rationales are more likely to explain the same permission group’s purposes in their descriptions than apps that do not provide rationales. We further discuss important implications from these findings.

Index Terms—Android Security, Runtime Permission, Rationale, Natural Language Processing

I. INTRODUCTION

Mobile security and privacy are two challenging tasks [1]–[7]. Recently user privacy issues gather tremendous attention after the Facebook-Cambridge Analytica data scandal [8]. Android’s current solution for protecting the users’ private data resources mainly relies on its sandbox mechanism and the permission system. Android permissions control the users’ private data resources, e.g., locations and contact lists. The permission system regulates an Android app to request permissions, and the app users must grant these permissions before the app can get access to the users’ sensitive data.

In earlier versions of Android, permissions are requested at the installation time. However, studies [3], [5] show that the install-time requests cannot effectively warn the users about potential security risks. The users are often not aware of the fact that permissions are requested, and the users also have poor understandings on the meanings and purposes of using the permissions [3], [9]. It is a critical task to educate the users by explaining permission purposes so that the users can better understand the purposes [5], [10], [11].

Since Android 6.0 (Marshmallow), the permission system has been replaced by a new system that requests permission

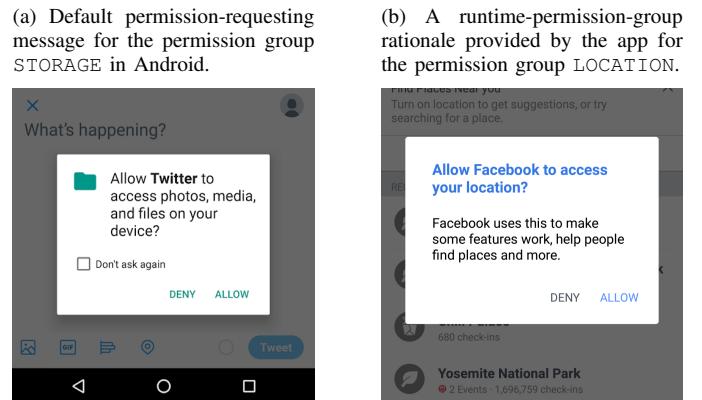


Fig. 1

groups [12] at runtime. An example of runtime-permission-group requests is in Figure 1a, where Android shows the default permission-requesting message for the permission group STORAGE¹. The runtime model has three advantages over the old model. (1) It gives the users more warnings than the install-time model. (2) It allows the users to control an app’s privileges at the permission-group level. (3) It gives apps the opportunity to embed their permission-group requests in contexts, so that the requests are self-explanatory. For example, in Figure 1a, a request for accessing the user’s gallery is prompted when she is about to send a Tweet.

With the runtime-permission system, each Android app can leverage a dialog to provide a customized message for explaining its unique purpose of using the permission group. In Figure 1b, we show an example of such messages from the *Facebook* app for explaining the purpose of requesting the user’s location: “*Facebook uses this to make some features work...*”. Such customized messages are called *runtime-permission-group rationales*. Runtime-permission-group rationales are often displayed before or after the permission-requesting messages, or upon the starting of the app. For the rest of this paper, for simplicity, whenever the context refers to a runtime-permission-group rationale or a runtime-

¹The permission-requesting message is the message displayed in the permission-requesting dialog (Figure 1a). For each permission group, this message is fixed across different apps. For example, the permission-requesting message for STORAGE is *Allow appname to access photos, media and files on your device?*

permission-group request, we use the term *rationale*, *runtime rationale*, and *permission-group rationale* in short for *runtime-permission-group rationale*; we use the term *permission request(-ing message)* in short for *runtime-permission-group request(-ing message)*.

There are three main reasons why runtime rationales are useful in the new permission system. (1) *Challenge in Explaining Background Purposes*. Although the runtime system allows permission-group requests to be self-explanatory in contexts, there exist cases where the permission groups are used in the background (e.g., read phone number, SMS) [13]. As a result, there does not exist a user-aware context for asking such permission groups. (2) *Challenge in Explaining non-Straightforward Purposes*. When the purpose of requesting a permission group is not straightforward, such as when the permission group is not for achieving a primary functionality, the context itself may not be clear enough to explain the purpose. For example, when the user is about to send a Tweet (Figure 1a), she may not notice that the location permission group is requested. (3) *Effectiveness of Natural Language Explanations*. Prior work [5] shows that the users find the usage of a permission better meets their expectation when the purpose of using such permission is explained with a natural language sentence. Furthermore, user studies [14] on Apple’s iOS runtime-permission system also demonstrate that displaying runtime rationales can effectively increase the users’ approval rates.

The effectiveness of explaining permission purposes relies on the contents of the explanation sentences [5]. Because the rationale sentences are created by apps, the quality of such rationales depends on how individual apps (developers) make decisions for providing rationales. Three essential decisions are (1) which permission group(s) the app should explain the purposes for; (2) for each permission group, what words should be used for explaining the permission group’s purpose; (3) how specific the explanation should be.

In this paper, we seek to answer the following questions: (1) what are the common decisions made by apps? (2) how are such decisions aligned with the goal of improving the users’ understanding of permission-group purposes? To understand the general patterns of apps’ permission-explaining behaviors, we conduct the first large-scale empirical study on runtime rationales. We collect an Android 6.0+ dataset consisting of 83,244 apps. From these apps, we obtain 115,558 rationale sentences. Our study focuses on the following five research questions.

RQ1: Overall Explanation Frequency. We investigate the overall frequency for apps to explain permission-group purposes with rationales. The result can help us understand whether the developers generally acknowledge the usefulness of runtime rationales, and whether the users are generally warned for the usages of different permission groups.

RQ2: Explanation Frequency for non-Straightforward vs. Straightforward Purposes. Prior work [5], [15] finds that the users have different expectations for different permission purposes. The Android official documentation [16] suggests

that apps provide rationales when the permission group’s purposes are not straightforward. Therefore, we investigate whether apps more frequently explain non-straightforward purposes than straightforward ones. The result can help us understand the helpfulness of rationales with the users’ understandings of permission-group purposes.

RQ3: Incorrect Rationales. We study the population of rationales where the stated purpose is different from the true purpose, i.e., the rationales are incorrect. Such study is related to user expectation, because incorrect rationales may confuse the users and mislead them into making wrong security decisions.

RQ4: Rationale Specificity. How exactly do apps explain purposes of requesting permission groups? How much information do rationales carry? Do rationales provide more information than the permission-requesting message? Do apps provide more specific rationales for non-straightforward purposes than for straightforward purposes?

RQ5: Rationales vs. App Descriptions. Are apps that provide rationales more likely to explain the same permission group’s purpose in the app description than apps that do not provide rationales? Are the behaviors of explaining a permission group’s purposes consistent in the app description and in rationales? Do more apps explain their permission-group purposes in the app description than in rationales?

The rest of this paper is organized as follows. Section II introduces background and related work, Section III describes the data collection process. Sections IV- VIII answer RQ1-RQ5. Sections IX- XI discuss threats to validity, implications, and conclusion of our study.

II. BACKGROUND AND RELATED WORK

Android Permissions and the Least-Privilege Principle. A previous study [2] shows that compared with attack-performing malware, a more prevalent problem in the Android platform is the *over-privilege* issue of Android permissions: apps often request more permissions than necessary. Felt *et al.* [3] evaluate 940 apps and find that one-third of them are over-privileged. Existing work leverages static-analysis techniques [2], [17] and dynamic-analysis techniques [1] to build tools for analyzing whether an app follows the *least-privilege principle*. The runtime-permission-group rationales we study are for helping the users make decisions on whether a permission-group request is over-privileged.

User Expectation. Over time, the research literature on Android privacy has focused on studying whether and how an app’s permission usage meets the users’ expectation [4], [5], [10], [18]–[23]. In particular, Lin *et al.* [5] find that the users’ security concern for a permission depends on whether they can expect the permission usage. Jing *et al.* [15] further find that even in the same app, the users have different expectations for different permissions. For example, in the *Skype* app, the users find the microphone permission more straightforward than the location permission. The Android official documentation [16] also points out this difference and suggests that app devel-

opers provide more runtime-permission-group rationales for purposes that are not straightforward to expect.

The research literature on user expectation can be categorized into three lines of work. The first line of work is on detecting contradictions between the code behavior and the user interface [18], [24]. The second line of work is on improving existing interfaces to enhance the users' awareness of permission usages [4], [13], [20]–[22], [25]. This line of work includes privacy nudging [4], access control gadget [22], and mapping between permissions and UI components [25]. In particular, Nissenbaum *et al.* [20] first propose the concept of privacy as the *contextual integrity*; i.e., the users' decision-making process for privacy relies on the contexts [13], [21], [26], [27]. The runtime-permission system incorporates the contextual integrity by allowing apps to ask for permission groups within the context. The third line of work is on using natural language sentences to represent or enhance the users' expectation regarding the permission usages [5], [10], [19], [28]. For example, Lin *et al.* [5] find that the users of an app are more comfortable with using the app when the app provides clarifications for the permission purposes than they do not provide such clarifications. Pandita *et al.* [10] further extract permission explaining sentences from app descriptions. Our study results presented in Section VIII show that apps explain purposes of requesting permission groups more frequently in the rationales than in the description.

Runtime Permission Groups and Runtime Rationales. Since the launch of the runtime-permission system, another line of work [5], [14], [29] (including our work) focuses on the runtime-permission system and the users' decisions on such system. In particular, Bonne *et al.* [29] conduct a study similar to the study by Lin *et al.* [5] under the runtime-permission system, showing the users' security decisions in the runtime system also rely on their expectations of the permission usages. The closest to our work is the study by Tan *et al.* [14] on the effects of runtime rationales in the iOS system. Their user-study results show that rationales can improve the users' approval rates for permission requests and increase the comfortableness for the users to use the app. Although they have not observed a significant correlation between the rationale contents and the approval rates, such observations may be due to the fact that only one fake app is examined with limited user feedback. As a result, such unrelatedness cannot be trivially generalized to our case. Wijesekera *et al.* [30] redesigns the timing of runtime prompts to reduce the *satisficing* and *habituation* issues [31]–[34]. Both Wijesekera *et al.* [30] and Olejnik *et al.* [35] leverage machine learning techniques to reduce user efforts in making decisions for permission requests.

III. DATA COLLECTION

A. Crawling Apps

Since the launch of Android 6.0, many apps have migrated to support the newer versions of Android. To obtain as many Android 6.0+ apps as possible, we crawl apps from the following two sources: (1) we crawl the top-500 apps in each category from the Google Play store, obtaining 23,779 apps in total; (2) we crawl 482,591 apps from APKPure [36], which is

another app store with copied apps (same ID, same category, same description, etc.) from the Google Play store². From the two sources, we collect 494,758 apps. Among these apps, we find 83,244 apps that (1) contain version(s) under Android 6.0+; (2) request at least 1 out of the 9 dangerous permission groups (Table I). We use these 83,244 apps as the dataset in this paper³.

B. Annotating Permission-group Rationales

For each app found in the preceding step, we annotate and extract runtime rationales from the app. Same as other static user interface texts, runtime rationales are stored in an app's `./res/values/strings.xml` file. Each line of this file contains a rationale's name and the content of the rationale.

The size of our dataset dictates that it is intractable to manually annotate all the string variables. As a result, we leverage two automatic sentence-annotating techniques: (1) keyword matching; (2) CNN sentence classifier. The automatic annotation is a two-step process.

Annotating Rationales for All Permission Groups. For the first step, we design a keyword matching technique to annotate whether a string variable contains mentions of a permission group. More specifically, we assign a binary label to each string variable by matching the variable's name or content against 18 keywords referring to permission groups, including “*permission*”, “*rationale*”, and “*toast*”⁴. To estimate the recall of keyword matching, we randomly sample 10 apps and inspect their string resource files. The result of our inspection shows that such keyword matching found all the rationales in the 10 apps.

Annotating Rationales for the 8 Dangerous Permission Groups⁵. For the second step, we use the CNN sentence classifier [38], [39] to annotate the outputs from the first step. The annotations indicate whether each rationale describes 1 of the 9 dangerous permission groups [12]. The 9 permission groups contain 26 permissions. These permission groups' protection levels are dangerous and the purposes of requesting these permission groups are relatively straightforward for the users to understand. For each permission group, we train a different CNN sentence classifier. We manually annotate 200~700 rationales as the training examples for each classifier. After applying CNN, we estimate the classifier's false positive rate (FP) and false negative rate (FN) by inspecting 100 output examples in each permission group. The average FP (FN) over the 8 permission groups is 5.1% (6.8%) and the maximum FP (FN) is 13% (16%). In total, CNN annotates 115,558 rationales, which can be found on our project's website [37].

²We are not able to collect all these apps from the Google Play store, due to its anti-theft protection that limits the downloading scale.

³To the best of our knowledge, this dataset is the largest app collection on runtime rationales; it is orders of magnitude larger than other runtime-rationale collections in existing work [13], [14].

⁴The complete list of the 18 keywords can be found on our project website [37].

⁵In this paper, we skip the BODY_SENSORS permission group because it contains too few rationales.

TABLE I: The number of the used apps (the #used apps column), the explained apps (the #explained apps column), and the proportion of explained app in the used apps (the %exp column). We sort the permission groups by #used apps.

permgroup	#used apps	#explain-ed apps	%exp	%exp (top)
STORAGE	73,031	14,668	20.2%	28.3%
LOCATION	32,648	7,088	21.6%	30.7%
PHONE	31,198	2,070	6.7%	11.0%
CONTACTS	23,492	2,607	11.1%	17.7%
CAMERA	16,557	4,235	25.6%	37.7%
MICROPHONE	9,130	2,152	23.5%	28.0%
SMS	4,589	589	12.8%	16.0%
CALENDAR	2,492	357	14.2%	22.6%
BODY_SENSORS	122	16	13.1%	15.4%
overall	83,244	19,879	23.8%	33.9%

Discussion. One caveat of our data collection process is that the rationales in string resource files are only *candidates* for runtime prompts. That is, they may not be displayed to the users. The reason why we do not study only the actually-displayed rationales is that such study relies on dynamic-analysis techniques, which limit the scale of our study subjects.

IV. RQ1: OVERALL EXPLANATION FREQUENCY

In the first step of our study, we investigate the proportion of apps that provide permission-group rationales to answer RQ1: how often do apps provide permission-group rationales? For each of the 9 permission groups, we count how many apps in our dataset request the permission group; we denote this value as #used apps. Among these apps, we further count how many of them explain the requested permission group's purposes with rationales; we denote this value as #explained apps. Given the two values, we measure the *explanation proportion* of a group of apps:

Definition 1 (Explanation proportion). *Given a group of apps, its explanation proportion of a permission group is the proportion of apps in that group to explain the purposes of requesting the permission group, i.e., #explained apps / #used apps. We denote the explanation proportion as %exp.*

In Table I, we show the values of #used apps, #explained apps, and %exp for each permission group. In addition, we compute the %exp value for only the categorical top-500 apps; we denote this value as %exp (top).

Result Analysis. From Table I we can observe three findings. (1) Overall, 23.8% apps provide runtime rationale. (2) The top-500 apps more frequently explain the purposes of using permission groups than the overall apps do. (3) The purposes of the four permission groups STORAGE, LOCATION, CAMERA, and MICROPHONE are more frequently explained than the other five permission groups.

Finding Summary for RQ1. 23.8% apps provide runtime rationales for their permission-group requests. Among all the permission groups, four groups' purposes are explained more often than the other permission groups. This result may imply

TABLE II: The app sets for measuring the correlation between the usage proportion and the explanation proportion. The apps in each set share the same purpose (the purpose column) to use the primary permission group (the permgroup column) with the usage proportion (the %use column).

appset	permgroup	purpose	%use	#apps
file mgr	STORAGE	file managing	95.4%	499
video players	STORAGE	store video	96.6%	1,306
photography	STORAGE	store photos	99.7%	3,534
maps&nnavi	LOCATION	GPS navigation	92.6%	1,541
weather	LOCATION	local weather	95.4%	908
travel&local	LOCATION	local search	87.8%	2,647
lockscreen	PHONE	answer call when screen locked	82.6%	425
voip call	PHONE	make calls	84.9%	847
caller id	PHONE	caller id	92.0%	175
caller id	CONTACTS	caller id	86.7%	196
mail	CONTACTS	auto complete	77.1%	140
contacts	CONTACTS	contacts backup	85.8%	259
flashlight	CAMERA	flashlight	96.6%	298
qrscan	CAMERA	qr scanner	88.4%	155
camera	CAMERA	selfie&camera	71.4%	749
recorder	MIC	voice recorder	75.7%	559
video chat	MIC	video chat	77.0%	139
sms	SMS	sms	60.4%	379
calendar	CALEND	calendar	36.0%	300

that app developers are less familiar with the purposes of PHONE and CONTACTS.

V. RQ2: EXPLANATION FREQUENCY FOR NON-STRAIGHTFORWARD VS. STRAIGHTFORWARD PURPOSES

In the second part of our study, we seek to *quantitatively* answer RQ2: do apps provide more rationales for non-straightforward permission-group purposes than for straightforward permission-group purposes?

It is challenging to *precisely* measure the straightforwardness for understanding the purpose of requesting a permission group. The reason for such challenge is that such straightforwardness relies on each user's existing knowledge, which varies from user to user. Therefore, we propose to *approximate* the straightforwardness by measuring the *usage proportion* of a permission group in a set of apps:

Definition 2 (Usage proportion). *Given a set of apps, its usage proportion (denoted as %use) of a permission group is the proportion of the apps (in this set) that request the permission group.*

Our approximation is based on the observation that if a permission group is frequently used by a set of apps, the permission-group purpose in that app set is often also straightforward to understand. For example, in a camera app, the users are more likely to understand the purpose of the camera permission group than the location permission group [16]; meanwhile, our statistics show that camera apps also more frequently request the camera permission group (71.4%) than the location permission group (27.0%).

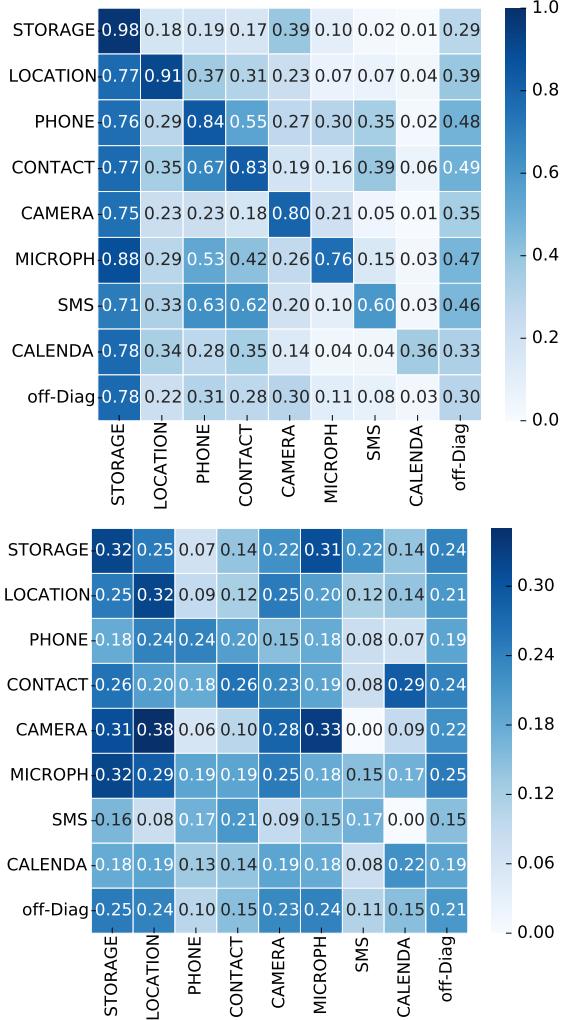


Fig. 2: The usage proportion (top) and the explanation proportion (bottom) of the app sets in Table II. Each element at (Q, P) shows the proportion of apps in set Q to use/explain the purpose of permission group P .

To answer RQ2, we first introduce the definitions of the primary permission group.

Definition 3 (Primary Permission Group). *Given a set of apps that share the same primary functionality, if any app relies on (does not rely on) requesting a permission group to achieve that primary functionality, we say that this permission group is a primary (non-primary) permission group to this app set, and this app set is a primary (non-primary) app set to this permission group. An example of such primary (non-primary) pairs is GPS navigation apps and LOCATION (CAMERA) permission group.*

To study the relation between the straightforwardness of permission-group purposes and explanation proportions, we leverage the following three-step process. (1) For each permission group P , we use keyword matching to identify 1~3 app sets such that P is a primary permission group to these app sets. (2) For each permission group Q , we merge its primary app sets to obtain a larger primary app set for Q . (3) For

TABLE III: The Pearson correlation tests of each permission group, between the usage proportion and the explanation proportion on the 35 Play-store app sets.

STORAGE	LOC	PHONE	CONTACT	CAMERA	MIC						
r	p	r	p	r	p	r	p	r	p	r	p
.4	8e-3	.6	1e-3	.5	6e-2	.8	1e-3	-.5	2e-2	.2	.5

each permission group P and the merged app sets for each permission group Q , we compute the proportion for app set Q to use/explain P , obtaining two 8×8 matrices. We show all the app sets in Table II, and the two matrices in Figure 2. In each matrix in Figure 2, each row corresponds to a merged app set Q and each column corresponds to a permission group P . For each row/column, we also compute the average over its off-diagonal elements and show these values in an additional column/row named off-Diag. That is, elements in off-Diag show the average over non-primary permission groups/app sets.

Why Using Primary Permission Groups? By introducing primary permission groups, we are able to identify permission-group purposes that are clearly straightforward (Table II), so that the boundaries between straightforward purposes and non-straightforward purposes are relatively well defined. We can observe such boundaries from the usage proportion matrix (Figure 2, top).

Result Analysis. We can observe the following findings from the explanation matrix in Figure 2 (bottom). (1) By comparing every diagonal element with its two off-Diag counterparts, we can observe that the diagonal elements are usually larger, indicating that straightforward permission-group purposes are explained more frequently than non-straightforward ones. On the other hand, there exist a few exceptional cases in LOCATION, MICROPHONE, SMS, and CALENDAR where at least one off-diagonal element is larger than the diagonal element, indicating that non-straightforward permission-group purposes are explained more frequently in these cases. (2) By comparing the elements in the off-Diag row, we find that the permission groups for which non-straightforward purposes are most explained are STORAGE, LOCATION, CAMERA, and MICROPHONE. Such result is consistent with the overall explanation proportions in Table I.

Measuring Correlation Over All Apps. Because the app sets in Table II cover only a subset of apps, we further design the second measurement study to capture all apps in our dataset. The second study includes the following two-step process. (1) Based on the app categories in the Google Play store, we partition all apps into 35 sets. After the partition, the two permission groups SMS and CALENDAR contain too few rationales in each app set, and therefore we discard these two permission groups. (2) For each permission group, we compute all its usage proportions and explanation proportions in the 35 app sets, and test the Pearson correlation coefficient [40] between the usage proportions and explanation proportions. In Table III, we show the results of the Pearson tests. We can observe that 4 out of the 6 tests show significantly positive correlation, i.e., straightforward purposes are usually more

frequently explained. Such results are generally consistent with the results in Figure 2.

Finding Summary for RQ2. Overall, apps *have not* provided more runtime rationales for non-straightforward permission-group purposes than for straightforward ones except for a few cases. This result implies that the majority of apps *have not* followed the suggestion from the Android official documentation [16] to provide rationales for non-straightforward permission-group purposes.

VI. RQ3: INCORRECT RATIONALES

In the third part of our study, we investigate the correctness of permission-group rationales. We seek to answer RQ3: does there exist a significant proportion of runtime rationales where the stated purposes do not match the true purposes?

It is challenging to derive an app’s true purpose for requesting a permission group. However, we can coarsely differentiate between purposes by checking the permissions under a permission group. Among the 9 permission groups in Android 6.0 and higher versions, 6 permission groups each contain more than one permission [12]. For example, the PHONE permission group controls the access to phone-call-related sensitive resources, and this permission group contains 9 phone-call-related permissions: CALL_PHONE, READ_CALL_LOG, READ_PHONE_STATE, etc. By examining whether the app requests READ_CALL_LOG or READ_PHONE_STATE, we can differentiate between the purposes of reading the user’s call logs and accessing the user’s phone number.

In order to easily identify the mismatches between the stated purpose and the true purpose, we study 3 permission groups consisting of relatively diverse permissions: PHONE, CONTACTS, and LOCATION. In particular, each of the 3 groups contains 1 permission such that 90% apps requesting the group have requested that permission (whereas other permissions in the same group are requested less frequently); therefore, we name such permission a *basic permission*. The basic permissions of PHONE, CONTACTS, and LOCATION are READ_PHONE_STATE, GET_ACCOUNTS, and ACCESS_COARSE_LOCATION, respectively.

Definition 4 (Apps with Incorrect Rationales). *We identify two cases for an app to contain incorrect rationale(s): (1) all the rationales state that the app requests only the basic permission, but in fact, the app has requested other permissions (in the same permission group); (2) the app requests only the basic permission, but it contains some rationales stating that it has requested other permissions (in the same permission group).*

How many apps does each of the two incorrect cases contains? Both cases can mislead the user to make wrong decisions. For case (1), the user may grant the permission-group request with the belief that she has granted only the basic permission, but in fact she has granted other permissions. For case (2), the user may deny the permission-group request, because the stated purpose of such permission group seems to be unrelated to the app’s functionality, e.g., when a music

TABLE IV: The upper table shows the criteria for annotating the basic permission and other permissions in the same permission group. The lower table shows the estimated lower bounds on the numbers of apps containing incorrectly stated rationales.

		CONTACTS	PHONE	LOCATION
annotate criterion	basic per-mission class (a)	google account/ sign in/ email add dress	pause incoming call/ imei/ identity/ number/ cellular	coarse loc /area/region /approximate /beacon /country
	other permissions class (b)	contacts/ friends/ phonebook	make call/ call phone/ call logs	driving/ fine loc/ coordinate
incorrect apps	case (1)	#err %err	#err %err	#err %err
	93	4.6	139	11.3
	case (2)	#err %err	#err %err	#err %err
	76	13.2	37	4.2
				3 0.6

player app requests the READ_PHONE_STATE permission only to pause the music when receiving phone calls, the rationale can raise the user’s security concern by stating that the music app needs to make a phone call. After the user denies the phone permission group, the app also loses the access to pausing the music.

To study the populations of the two preceding incorrect cases, we again leverage the aforementioned CNN sentence classifier [38]. We classify each runtime rationale into one of the following three classes: (a) the rationale states the purpose of requesting a basic permission; (b) the rationale states the purpose of requesting a permission other than the basic permission; (c) neither (a) nor (b). For each of the three permission groups, we manually annotate 600~900 rationales as the training data. After we obtain the predicted labels, we manually judge the resulting rationales that are predicted as (a) or (b) to make sure that there do not exist false positive annotations for incorrect case (1) or (2). In Table IV, we show the lower-bound estimations (#err and %err) of the two incorrect cases’ populations. We also show the detailed criteria of our annotations for (a) and (b). The list of incorrect rationales and their apps can be found on our project website [37].

Result Analysis. From Table IV we can observe that there exist a significant proportion of incorrectly stated runtime rationales, especially in the incorrect case (1) of the phone permission group and the incorrect case (2) of the contacts permission group. In contrast, there exist fewer incorrect cases in the location permission group. The reason for the location permission group to contain fewer incorrect cases may be that the majority of apps claim only the usage of location, without specifying whether the requested location is fine or coarse. The contacts and phone permission groups contain more diverse purposes than the location group does, and our study results show that a significant proportion of apps requesting the two groups state the wrong purposes. For example, a significant number of FM radio apps state in the rationales that these apps *only* need to use the phone state to pause the radio when receiving incoming calls; however, these apps have also

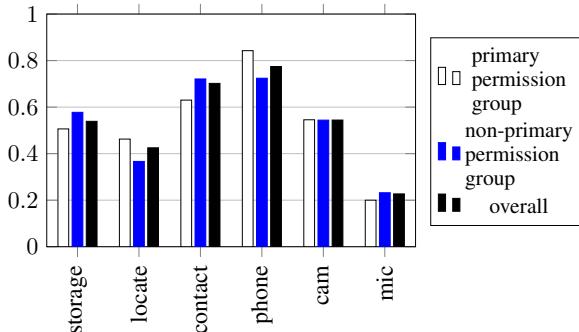


Fig. 3: The proportions of non-redundant rationales.

requested the CALL_PHONE permission, indicating that if the user grants the permission group, these apps also gain the access to *making phone calls* within the app.

Finding Summary for RQ3. There exist a significant proportion of incorrect runtime rationales for the CONTACTS and the PHONE permission groups. This result implies that apps may have confused the users by stating the incorrect permission-group purposes for PHONE and CONTACTS.

VII. RQ4: RATIONALE SPECIFICITY

In the fourth part of our study, we look into the informativeness of runtime rationales. In particular, we seek to answer RQ4: do rationales (e.g., the rationale in Figure 1b) provide more specific information than the system-provided permission-requesting messages (e.g., the message in Figure 1a)?

Definition 5 (Redundant Rationales). *If a runtime rationale states only the fact that the app is requesting the permission group, i.e., it does not provide more information than the permission-requesting message, we say that the rationale is redundant, and otherwise non-redundant.*

Among all the runtime rationales, how many are non-redundant ones? How much do the proportions of non-redundant rationales in each permission group vary across permission groups?

To study the population of non-redundant rationales, we leverage the named entity tagging (NER) technique [41]. The reason for us to leverage the NER technique is our observation that non-redundant rationales usually use some words to state the more specific purposes than the fact of using the permission group. Moreover, these purpose-stating words usually appear in textual patterns. As a result, we can leverage such textual patterns to detect non-redundant rationales. For example, in the following rationale, the words tagged with “S” explain the *specific* purpose of using the permission group PHONE, and the words tagged with _O are other words: “*this_O radio_O application_O would_O like_O to_O use_O the_O phone_O permission_O to_S pause_S the_S radio_S when_S receiving_S incoming_S calls_S*”. We train a different NER tagger for each of the top-6 permission groups in Table I⁶. For each permission group, we manually annotate

⁶We skip SMS and CALENDAR, because they both contain too few rationales for estimating the proportions of non-redundant rationales.

200~1,000 training examples. To evaluate the performance of our NER tagger, we randomly sample 100 rationales from NER’s output for each permission group, and manually judge these sampled rationales. Our judgment results show that NER’s prediction accuracy ranges from 85% to 94%. The lists of redundant and non-redundant rationales tagged by NER can be found on our project website [37]. Next, we obtain the proportions of non-redundant rationales in each permission group. We plot these proportions in Figure 3.

Result Analysis. We can observe three findings from Figure 3 and additional experiments. (1) The proportions of redundant runtime rationales range from 23% to 77%. (2) While the two permission groups PHONE and CONTACTS have the lowest explanation proportions (Figure 2), they have the highest non-redundant proportions. The reason why most phone and contacts rationales are non-redundant is that they usually specify whether the permission group is used for the basic permission or other permissions. (3) We also study the proportions of non-redundant rationales in the app sets defined in Table II, but we have not observed a significant correlation between the usage proportions and the non-redundant proportions.

Finding Summary for RQ4. A large proportion of the runtime rationales have not provided more specific information than the permission-requesting messages. The rationales in PHONE and CONTACTS are most likely to explain more specific purposes than the permission-requesting messages. This result implies that a large proportion of the rationales are either unnecessary or should be more specifically explained.

VIII. RQ5: RATIONALES VS. APP DESCRIPTIONS

In the fifth part of our study, we look into the correlation between the runtime rationales and the app description. We seek to answer RQ5: how does explaining a permission group’s purposes in the runtime rationales relate to explaining the same permission group’s purposes in the app description? Are apps that provide rationales more likely to explain the same permission group’s purposes in the app description than apps that do not provide rationales?

To identify apps that explain the permission-group purposes in the description, we leverage the WHYPER tool and the keyword matching technique [10]. WHYPER is a state-of-the-art tool for identifying permission-explaining sentences. We apply WHYPER on the CONTACTS and the MICROPHONE permission groups. Because WHYPER [42] does not provide the entire pipeline solution for other frequent permission groups, we use the keyword matching technique to match sentences for another permission group LOCATION. Prior work [11] also leverages keyword matching for efficient processing. We show the results in Table V.

Result Analysis. From Table V, we can observe two findings. (1) In two out of the three cases, the correlations are significantly positive. Therefore, an app that provides runtime rationales is also more likely to explain the same permission group’s purpose in the description. (2) There exist

TABLE V: The number of apps that explain a permission group’s purposes in the app description (the #apps *descript* column), in the rationales (the #apps *rationales* column), in both (the #apps *both* column), and the Pearson correlation coefficients between whether an app explains a permission group’s purpose in the description vs. rationales (the Pearson column).

	#apps descript	#apps rationales	#apps both	Pearson
LOCATION	5,747	7,088	2,028	(0.15, 1.86e-168)
CONTACTS	1,542	2,607	394	(0.12, 1.5e-78)
MICROPH	957	2,152	245	(0.02, 0.12)

more apps using runtime rationales to explain the permission-group purposes than apps that use the descriptions.

Finding Summary for RQ5. The explanation behaviors in the description and in the runtime rationales are often positively correlated. Moreover, more apps use runtime rationales to explain purposes of requesting permission groups than using the descriptions. This result implies that apps’ behaviors of explaining permission-group purposes are generally consistent across the descriptions and the rationales.

IX. THREATS TO VALIDITY

The threats to external validity primarily include the degree to which the studied Android apps or their runtime rationales are representative of true practice. We collect the Android apps from two major sources, one of which is the Google Play store, the most popular Android app store. Such threats could be reduced by more studies on more Android app stores in future work. The threats to internal validity are instrumentation effects that can bias our results. Faults in the used third-party tools or libraries might cause such effects. To reduce these threats, we manually double check the results on dozens of Android apps under analysis. Human errors during the inspection of data annotations might also cause such effects. To reduce these threats, at least two authors of this paper independently conduct the inspection, and then compare the inspection results and discuss to reach a consensus if there is any result discrepancy.

X. IMPLICATIONS

In this paper, we attain multiple findings for Android runtime rationales. These findings imply that developers may be less familiar with the purposes of the PHONE and CONTACTS permission groups and some rationales in these groups may be misleading (RQ1 and RQ3); the majority of apps have not followed the suggestion for explaining non-straightforward purposes [16] (RQ2); a large proportion of rationales may either be unnecessary or need further details (RQ4); and apps’ explanation behaviors are generally consistent across the descriptions and the rationales (RQ5). Such findings suggest that the rationales in existing apps may not be optimized for the goal of improving the users’ understanding of permission-group purposes. Based on these implications, we propose two suggestions on the system design of the Android platform.

Official Guidelines or Recommender Systems. It is desirable to offer an official guideline or a recommender system for suggesting which permission-group purposes to explain [11], e.g., on the official Android documentation or embedded in the IDE. For example, such recommender system can provide a list of functionalities, so that the developer can select which functionalities are used by the app. Based on the developer’s selections, the system scans the permission-group requests by the app, and lets the developer know which permission group(s)’ purposes may look non-straightforward to the users. In addition, the system can suggest rationales for the developers to adapt or to adopt [11].

Controls over Permissions for the Users. When a permission group contains multiple permissions, such design increases the challenges and errors in explaining the purposes of requesting such permission group. It is interesting to study whether a user actually knows which permission she has granted, e.g., does a weather app use her precise location or not? One potential approach to improve the users’ understanding of permission-group purposes is to further scale down the permission-control granularity from the user’s end. For example, the “permission setting” in the Android system can display a list showing whether each of the user’s *permissions* (instead of permission groups) has been granted; and doing so also gives the users the right to revoke each permission individually.

XI. CONCLUSION

In this paper, we have conducted the first large-scale empirical study on runtime-permission-group rationales. We have leveraged statistical analysis for producing five new findings. (1) Less than one-fourth of the apps provide rationales; the purposes of using PHONE and CONTACTS are the least explained. (2) In most cases, apps explain straightforward permission-group purposes more than non-straightforward ones. (3) Two permission groups PHONE and CONTACTS contain significant proportions of incorrect rationales. (4) A large proportion of the rationales do not provide more information than the permission-requesting messages. (5) Apps’ explanation behaviors in the rationales and in the descriptions are positively correlated. Our findings indicate that developers may need further guidance on which permission groups to explain the purposes and how to explain the purposes. It may also be helpful to grant the users controls over each permission.

Our study focuses on analyzing natural language rationales. Besides the rationales, other UI components (e.g., layout, images/icons, font size) can also affect the users’ decision making. In future work, we plan to study the effects of runtime-permission-group requests when considering these factors, and study ways to encourage the developers to provide higher-quality warnings than the current ones.

Acknowledgment. We thank the anonymous reviewers and Xiaofeng Wang for their useful suggestions. This work was supported in part by NSF CNS-1513939, CNS-1408944, CCF-1409423, and CNS-1564274.

REFERENCES

- [1] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. D. McDaniel, and A. N. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*. ACM, 2014, pp. 393–407.
- [2] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the ACM Conference on Computer and Communications security*. ACM, 2011, pp. 627–638.
- [3] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Proceedings of the Symposium on Usable Privacy and Security*. USENIX Association, 2012, pp. 3:1–3:14.
- [4] H. Almuhimedi, F. Schaub, N. M. Sadeh, I. Adjerid, A. Acquisti, J. Gluck, L. F. Cranor, and Y. Agarwal, "Your location has been shared 5,398 times! A field study on mobile app privacy nudging," in *Proceedings of the Annual ACM Conference on Human Factors in Computing Systems*. ACM, 2015, pp. 787–796.
- [5] J. Lin, N. M. Sadeh, S. Amini, J. Lindqvist, J. I. Hong, and J. Zhang, "Expectation and purpose: Understanding users' mental models of mobile app privacy through crowdsourcing," in *Proceedings of the ACM Conference on Ubiquitous Computing*. ACM, 2012, pp. 501–510.
- [6] J. Lin, B. Liu, N. M. Sadeh, and J. I. Hong, "Modeling users' mobile app privacy preferences: Restoring usability in a sea of permission settings," in *Proceedings of the Symposium on Usable Privacy and Security*. USENIX Association, 2014, pp. 199–212.
- [7] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "AppContext: Differentiating malicious and benign mobile app behaviors using context," in *Proceedings of the International Conference on Software Engineering*. IEEE Computer Society, 2015, pp. 303–313.
- [8] "Facebook and cambridge analytical data breach," https://en.wikipedia.org/wiki/Facebook_and_Cambridge_Analytica_data_breach, accessed: 2018-07-27.
- [9] P. G. Kelley, S. Consolvo, L. F. Cranor, J. Jung, N. M. Sadeh, and D. Wetherall, "A conundrum of permissions: Installing applications on an Android smartphone," in *Financial Cryptography Workshops*. Springer, 2012, pp. 68–79.
- [10] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "WHYPER: Towards automating risk assessment of mobile applications," in *Proceedings of the USENIX Security Symposium*. USENIX Association, 2013, pp. 527–542.
- [11] X. Liu, Y. Leng, W. Yang, C. Zhai, and T. Xie, "Mining Android app descriptions for permission requirements recommendation," in *Proceedings of the International Requirements Engineering Conference*. IEEE Computer Society, 2018.
- [12] "Android permission groups," <https://developer.android.com/guide/topics/permissions/requesting.html#perm-groups>, 2018, accessed: 2018-07-27.
- [13] K. K. Micinski, D. Votipka, R. Stevens, N. Kofinas, M. L. Mazurek, and J. S. Foster, "User interactions and permission use on Android," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2017, pp. 362–373.
- [14] J. Tan, K. Nguyen, M. Theodorides, H. Negrón-Arroyo, C. Thompson, S. Egelman, and D. A. Wagner, "The effect of developer-specified explanations for permission requests on smartphone user behavior," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2014, pp. 91–100.
- [15] Y. Jing, G.-J. Ahn, Z. Zhao, and H. Hu, "RiskMon: Continuous and automated risk assessment of mobile applications," in *Proceedings of the ACM Conference on Data and Application Security and Privacy*. ACM, 2014, pp. 99–110.
- [16] "Should show request permission rationale API," [https://developer.android.com/reference/android/support/v4/app/ActivityCompat#shouldShowRequestPermissionRationale\(android.app.Activity.java.lang.String\)](https://developer.android.com/reference/android/support/v4/app/ActivityCompat#shouldShowRequestPermissionRationale(android.app.Activity.java.lang.String)), 2018, accessed: 2018-07-27.
- [17] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android permission specification," in *Proceedings of the ACM Conference on Computer and Communications Security*. ACM, 2012, pp. 217–228.
- [18] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "AsDroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction," in *Proceedings of the International Conference on Software Engineering*. ACM, 2014, pp. 1036–1046.
- [19] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proceedings of the International Conference on Software Engineering*. ACM, 2014, pp. 1025–1035.
- [20] H. Nissenbaum, "Privacy as contextual integrity," Washington University School of Law, 2004, pp. 101–139.
- [21] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. A. Wagner, and K. Beznosov, "Android permissions remystified: A field study on contextual integrity," in *Proceedings of the USENIX Security Symposium*. USENIX Association, 2015, pp. 499–514.
- [22] F. Roesner, T. Kohno, A. Moschuk, B. Parno, H. J. Wang, and C. Cowan, "User-driven access control: Rethinking permission granting in modern operating systems," in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2012, pp. 224–238.
- [23] P. G. Kelley, L. F. Cranor, and N. M. Sadeh, "Privacy as part of the app decision-making process," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2013, pp. 3393–3402.
- [24] B. Andow, A. Acharya, D. Li, W. Enck, K. Singh, and T. Xie, "UiRef: Analysis of sensitive user inputs in Android applications," in *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks*. ACM, 2017, pp. 23–34.
- [25] Y. Li, Y. Guo, and X. Chen, "PERUIM: Understanding mobile application privacy with permission-UI mapping," in *Proceedings of the ACM Conference on Ubiquitous Computing*. ACM, 2016, pp. 682–693.
- [26] K. Z. Chen, N. M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. R. Magrino, E. X. Wu, M. Rinard, and D. X. Song, "Contextual policy enforcement in Android applications with permission event graphs," in *Proceedings of the Network & Distributed System Security Symposium*. The Internet Society, 2013.
- [27] D. Votipka, K. Micinski, S. M. Rabin, T. Gilray, M. M. Mazurek, and J. S. Foster, "User comfort with Android background resource accesses in different contexts," in *Proceedings of the Symposium on Usable Privacy and Security*. USENIX Association, 2018.
- [28] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "AutoCog: Measuring the description-to-permission fidelity in Android applications," in *Proceedings of the ACM Conference on Computer and Communications Security*. ACM, 2014, pp. 1354–1365.
- [29] B. Bonné, S. T. Peddinti, I. Bilogrevic, and N. Taft, "Exploring decision-making with Android's runtime permission dialogs using in-context surveys," in *Proceedings of the Symposium on Usable Privacy and Security*. USENIX Association, 2017, pp. 195–210.
- [30] P. Wijesekera, A. Baokar, L. Tsai, J. Reardon, S. Egelman, D. Wagner, and K. Beznosov, "The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences," in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2017, pp. 1077–1093.
- [31] D. Akhawe, B. Amann, M. Vallentin, and R. Sommer, "Here's my cert, so trust me, maybe?: Understanding TLS errors on the web," in *Proceedings of the International Conference on World Wide Web*. ACM, 2013, pp. 59–70.
- [32] M. S. Wogalter, V. C. Conzola, and T. L. Smith-Jackson, "Research-based guidelines for warning design and evaluation," vol. 33, no. 3. Elsevier, 2002, pp. 219–230.
- [33] M. Harbach, S. Fahl, P. Yakovleva, and M. Smith, "Sorry, I don't get it: An analysis of warning message texts," in *Proceedings of the International Conference on Financial Cryptography and Data Security*. Springer, 2013, pp. 94–111.
- [34] F. Schaub, R. Balebako, A. L. Durity, and L. F. Cranor, "A design space for effective privacy notices," in *Proceedings of the Symposium On Usable Privacy and Security*. USENIX Association, 2015, pp. 1–17.
- [35] K. Olejnik, I. Dacosta, J. S. Machado, K. Huguenin, M. E. Khan, and J.-P. Hubaux, "SmarPer: Context-aware and automatic runtime-permissions for mobile devices," in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2017, pp. 1058–1076.
- [36] "APKPure website," <https://www.apkpure.com>, 2018, accessed: 2018-07-27.
- [37] "Runtime permission rationale project website," <https://sites.google.com/view/runtimepermissionproject/>, accessed: 2018-07-27.
- [38] "A tensorflow implementation of CNN text classification," <https://github.com/dennybritz/cnn-text-classification-tf>, 2018, accessed: 2018-07-27.
- [39] Y. Kim, "Convolutional neural networks for sentence classification," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2014, pp. 1746–1751.
- [40] "Pearson correlation coefficient," https://en.wikipedia.org/wiki/Pearson_correlation_coefficient, 2018, accessed: 2018-07-27.

- [41] J. R. Finkel, T. Grenager, and C. D. Manning, “Incorporating non-local information into information extraction systems by Gibbs sampling,” in *Proceedings of the Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 2005, pp. 363–370.
- [42] “WHYPER tool,” <https://github.com/rahulpandita/Whyper>, accessed: 2018-07-27.

Interactions for Untangling Messy History in a Computational Notebook

Mary Beth Kery

Human-Computer Interaction Institute, CMU
mkery@cs.cmu.edu

Brad A. Myers

Human-Computer Interaction Institute, CMU
bam@cs.cmu.edu

Abstract—Experimentation through code is central to data scientists’ work. Prior work has identified the need for interaction techniques for quickly exploring multiple versions of the code and the associated outputs. Yet previous approaches that provide history information have been challenging to scale: real use produces a high number of versions of different code and non-code artifacts with dependency relationships and a convoluted mix of different analysis intents. Prior work has found that navigating these records to pick out the *relevant* information for a given task is difficult and time consuming. We introduce Verdant, a new system with a novel versioning model to support fast retrieval and sensemaking of messy version data. Verdant provides light-weight interactions for comparing, replaying, and tracing relationships among many versions of different code and non-code artifacts in the editor. We implemented Verdant into Jupyter Notebooks, and validated the usability of Verdant’s interactions through a usability study.

Keywords—exploratory programming, versioning, data science

I. INTRODUCTION

In data science, exploratory programming is essential to determining which data manipulations yield the best results [1] [2], [3]. It can be highly helpful to record what iterations were run under what conditions and under what assumptions about the data. This gives data scientists better certainty in their work, the ability to reproduce it, and a more effective understanding about where to focus their efforts next. Today, most of this experimental history is lost. Our studies [4] [5], as well as those by Rule et al [6] and a 2015 survey from Jupyter of over 1000 data science users [7] have all found task needs as well as strong direct requests from data scientists for improved version support.

In terms of versioning, where does data science programming diverge from any other form of code development? Typically in regular code development, the primary artifact that a programmer works with is code [8]. Data science programming relies on working with a broader range of artifacts: the code itself, important details within the code [4], parameters or data used to run the code [9], visualizations, tables, and text output from the code, as well as

	IncidentNum	Category	
0	150098210	NON-CRIMINAL	LOST PROPERTY
1	150098210	ROBBERY	ROBBERY, BODILY FORCE
2	150098210	ASSAULT	AGGRAVATED ASSAULT WITH BODILY F
3	150098210	SECONDARY CODES	DOMESTIC VIOLENCE
4	150098226	VANDALISM	MALICIOUS MISCHIEF, VANDALISM OF VE

Figure 1. Verdant in-line history interactions. For the top code cell, a ribbon visualization shows the versions of the third line of code. In the output cell below, a margin indicator on the right shows that there are 5 versions of the output.

notes the data scientist jots down during their experimentation [10]. The conditions under which code was run and under which data was processed gives meaning to a version of code [9]. Data scientists need to ask questions that require knowledge of history about specific artifacts, specific code snippets, and the relationships among those artifacts over time: “What code on what data produced this graph?”, “What was the performance of this model under these assumptions?”, “How did this code perform on this dataset versus this other dataset?”, etc. Seeing relationships among artifacts allows a data scientist to answer cause-and-effect questions and evaluate the results and the progress of their experimentation.

To achieve this level of history support, we aim to A) store a rich relational history for all artifacts, B) allow data scientists to pull out history specifically relevant to a given task, and C) clearly communicate how versions of different artifacts have combined together during experimentation.

Several related areas of tooling offer promising avenues towards these goals. Computational notebook development environments, such as Jupyter notebooks, have become highly popular for data science programming because a notebook allows a data scientist to see all their input, output, formatted notes, and code artifacts in one place, and thus more easily work with and communicate context [11]. Meanwhile, our prior research prototype called Variolite demonstrated several in-editor interactions for creating and manipulating versions of specific code snippets [4]. Only working with code snippets,

Variolite did not treat the issue of a mix of code and non-code artifacts or issues of scalability, thus further exploration of this form of lightweight in-editor interactions is needed to adapt these ideas to more complex situations. Finally, the field of provenance research, meaning “origin or history of ownership” [12], has argued for and developed methods for automatically collecting input, code, and output each time a programmer runs their code, in order to capture a complete history [13], [14]. Currently the best solutions available to data scientists are manually making Git commits at very frequent intervals, manually making copies of their code files, or manually writing logging code for parameter and output artifacts they want to record [4]. Besides lessening the burden on the programmer to manually version their artifacts, automated approaches can detect and store dependency relationships among artifacts [13].

Unfortunately, just collecting the appropriate history data is not enough. Prior provenance research illustrates that in real use, capturing history data produces a large number of versions with complex dependency relationships and a convoluted mix of different analysis intents that can become overwhelming for a human to interpret [13]. Behavioral research has found that it is both a challenging and tedious task for human programmers to pick out and adapt relevant version data from long logs of code history [15]. Even when using standard version control like Git, software developers often struggle with information overload from many versions, all of which are rarely labeled or organized in a clear enough way to easily navigate [8].

In this work, we explore the design space of new interactions for providing easy-to-use history support for data scientists in their day-to-day tasks. Untangling messy history logs to deliver them in a useful form requires both advances in how edit history is modeled, and active testing of potential user interactions on actual log data from realistic data science tasks. To facilitate this, we developed a prototype tool called *Verdant* (from the meaning “an abundance of growing plants” [16]) as an extension for Jupyter notebooks. By relying on existing Jupyter interactions to display code and non-code artifacts, Verdant adds a layer of history interactions on top of Jupyter’s interactions that are likely to be familiar to data scientists and already have been established to be usable even to novices [17]. Underlying Verdant, we develop a novel approach to version collection to model versions of all artifacts in the notebook along with dependency relationships among them. Using this gathered history data, we then explore the design space of lightweight interactions for:

1. Quickly retrieving versions of a specific artifact out of an abundance of versions of the entire document.
2. Comparing multiple versions of different artifacts including code, tables, and images, which benefit from different diff-ing techniques.

3. Walking the data scientist through how to reproduce a specific version of an artifact.

Finally, we validate the real life task-fit of these interactions in an initial usability study with five experienced data science programmers. All participants were successfully able to complete small tasks using the tool and discussed use cases for Verdant specific to their own day-to-day work. With feedback from these use case walkthroughs from participants, we discuss next steps in his design space.

II. RELATED WORK

Computational Notebooks: Computational notebook programming dates back to early ideas of “literate programming” by Knuth [18] in 1984. Although there are many examples today of computational notebooks like Databricks [19] or Colab [20], Jupyter is a highly popular and representative example with millions of users. Therefore, we chose to use it in this work, particularly since it is open-source and thus easy to extend. Computational notebooks show many different artifacts together in-line. Each artifact, like code or markdown, has its own “cell” in the notebook, and the programmer is free to execute individual cells in any order, thus avoiding needing to re-run computationally expensive steps. The cell structure is an important consideration for versioning tools. Since the cell is a discrete structure, it can be tempting to version a notebook by cells so that the user can browse all history specific to one cell. However, we caution against overly relying on cell structure, because prior behavioral work [5], [6] shows that notebook users commonly add lots of new cells, then reduce or recombine them into different cell structures as they iterate. Users also reorder and move around cells [5], [6]. Finally, Jupyter notebooks support “magic” commands, which are commands that start with “%” that a user can run in the notebook environment to inspect the environment itself. This includes a `%history` command that outputs a list of all code run in the current session. While prior history work in Jupyter notebooks [13] has relied on `%history`, we take a different approach since this `%history` prints only the plaintext code run in a tangle of different analysis tasks, and we aim to collect more specific context across all artifacts involved.

Provenance work: Provenance, tracking how a result was produced, has many different levels of granularity, all the way down to the operating system-level of the runtime environment [14]. In this work, we do not collect *absolute* provenance, since we only collect reasonably fine-grained runtime information about code, input, and output that is accessible from inside the computational notebook environment. The focus of our research is how to make provenance data *usable* to data scientists, and thus we focus on recording the history metadata most useful for data scientists at the cost of some precision. Pimentel et al. in 2015 created an extension to Jupyter notebooks that collects Abstract Syntax Tree (AST) information to record the execution order

and the function calls used to produce a result [13]. However, to retrieve this history, users must write SQL or Prolog queries into their notebook to retrieve either a list of metadata or a graph visualization of the resulting dependencies [13]. Instead of having users write more code to retrieve history, our focus in this work is to provide direct manipulation interactions which require far less skill from the user. Extensive prior provenance work has used graph visualizations to communicate provenance relationships to users [21], however graph visualizations are well known to be difficult for end-users to use [22], thus we avoid them.

Version History Interaction Techniques: In standard code versioning tools like Git, versions are shown as a list of commits, or a tree visualization to show different branches in a series of commits [23]. In tools like R Studio [24] a data scientist can see a list of code they have run so far. However just like Jupyter’s `%history` list, a list of code lacks any context to tell which code went with which analysis tasks or artifact context like input/outputs/notes needed to return to a prior version. Variolite tackles more specific version context by structuring in tool form the informal copy-paste versioning that data scientists already use [4]. In Variolite, programmers are able to select a little section of code, even just a line or a parameter, and wrap it in a “variant box” so that within that box, they can switch among multiple versions. Rather than shifting through full versions of the whole file, the programmer has the code variants that are meaningful to them directly in the editor. However, Variolite did not provide any support for non-code artifacts, and was highly limited by the manually drawn variant box. Variolite only recorded snippet-specific history inside the variant box, so the user could not move code in and out of the box without losing history. If a user did not think to put a variant box around everything of interest *before* running code experiments, it was not possible to recover snippet-specific history later. To avoid these limitations, our new Verdant system automatically collects all history so that a data scientist can flexibly inspect different parts of their work and always have access to its history data. Finally, prior work for fine-grained selective undo of code has collected versioning on a token-by-token level and visualized this through in-editor menus and an editor pane displaying a timeline [25]. Token-level edits are not terribly appropriate for data scientists because during experimentation, data scientists are more concerned with semantically-meaningful units of code like a parameter or method, rather than low level syntax edits [3].

Behavioral work on navigating versions: Navigating corpuses of version data and reusing bits of older versions has been shown to be difficult for programmers, from professional software engineers to novices [8], [15]. Srinivasa Ragavan et al. have modeled how programmers navigate through prior versions using Information Foraging Theory (IFT) [15] in which a programmer searches for the information by following clues called “scents”. Scents include features of a

version like its timestamp, output, and different snippets found within the code. To investigate how data science programmers specifically mentally formulate what aspect of a prior version they are looking for, we ran a brief survey with 45 participants [5]. We found that data scientists recall their work through many aspects like libraries used, visual aspects of graphs, parameters, results, and code, not all of which are easily expressed a textual search query [5]. Given these findings, we aim to support foraging and associative memory by providing plenty of avenues for a data scientist to navigate back to an experiment version based on whatever tidbit or artifact attribute they recall.

III. VERDANT VERSION MODEL

Verdant is built as an Electron app that runs a Jupyter notebook, and is implemented in HTML/CSS and Node.js. Although the interactions of Verdant are language-agnostic, since the implementation relies on parsing and AST models for code versioning, Verdant relies on a language-specific parser. We chose to support Python in this prototype, as it is a popular data science language. By substituting in a different parser, Verdant can work for any language.

Verdant uses existing means in Jupyter for displaying different types of media in order to capture versions of all artifacts in the notebook. For a single version of the notebook, (a “commit” using Git terminology), the notebook is captured in a tree structure. The root node of the tree is the notebook itself, and each cell in the notebook is a child node of the notebook. For code cells, their nodes are broken down further into versions by their abstract syntax tree (AST) structure, such that each syntactically meaningful span of text in the code can be recorded with its own versions. For output, markdown, and other multimedia cells, the cell is a node with no children, which means that a programmer can see versions of the output cell as a whole, but not of pieces of output.

A full version of the notebook is captured each time any cell is run. For efficiency, commits only create new nodes for whatever has changed, using reference pointers to all of the child nodes of the previous commit for whatever is the same. Versioning in this tree structure and at the AST level addresses many concerns of scale. For instance, imagine a data scientist Lucy has iterated on code for 257 different runs, but has only changed a certain parameter 3 times. Through AST versioning, Lucy does not have to sift through all 257 versions of her code with repetitive parameter values, but can instead simply retrieve the 3 unique versions of that parameter. Although AST versioning provides a great deal of flexibility to provide context-specific history, like Lucy’s 3 versions of her parameter, it adds algorithmic challenges. Namely, each time Lucy runs her code, there is the full version A of the AST which is the last recorded version of the program and a new full version B of the AST that is the result of all of Lucy’s new changes up to the point of the run. Matching two ASTs has been done previously, using heuristics like string-distance,

type and tree structure properties [26][27], however note that this matching has not been used in user-facing edit tracking before. Further, what is a correct match from a pure program structure perspective may not always match what is “correct” to the user. For instance, if Lucy changes a parameter 3 to total (“Main St.”), Lucy may want to see the history of these two AST nodes matched, since both are serving the exact same role as her parameter, however since these are far in both type and string-distance, a traditional matching algorithm would *not* match the two. Refining this matching algorithm to match user expectation is an area for future work, thus for the purposes of the immediate design exploration, Verdant implements a simple Levenshtein string matching algorithm: if the token edit distance between two AST nodes is less than or equal to 30% of the length of the nodes, Verdant considers them a match.

To collect dependency relationships, we run the Jupyter magics command %whos, which returns information from the running python kernel on the names and values of variables currently present in the notebook’s global environment. When one of these global variables changes value, we record which code cell ran immediately before the variable change, to approximate which code cell set the value of that variable, consistent with some prior code execution recording work [28]. For each code node that Verdant versions, Verdant inspects the code’s AST structure to identify which if any of the global variables that code snippet uses. If the code snippet uses a global variable, then a dependency is recorded between the code node and that specific version of the global variable, including which other code version produced the used value of the variable.

V. INTERACTIONS FOR VERSION FORAGING

Although a notebook may contain many code, output, and markdown cells, prior work suggests that data scientist work on only a small region of cells at a time for a particular exploration [5]. First, we show how Verdant uses inline interactions so that users can see versions of the task-related artifacts they are interested in, and not be overloaded with unrelated version information for the rest of the notebook.

A. Ambient Indicators

Following tried and tested [29] usability conventions of other tools that support investigating properties of code, like “linters,” a version tool should be non-disruptive while the user is focused on other tasks, while giving some ambient indication of what information is available to investigate further. Linters often use squiggly lines under code and indicator symbols in the margins next to the line of code the warning references. Verdant takes the approach in Figure 2: (A) no version information appears when a data scientist is reading through their notebook, but (B) when data scientists click on a cell to start working with it, they see an indicator in the margin that gives the number of versions of that cell (in this case 10). If the data scientist selects different spans of

code, the indicator changes height and label to show the number of unique versions of the selected code (in this case 9) (C). While a linter conventionally puts an icon on one line, we decided instead for the height of the version indicator to stretch from the bottom to the top of the text span it is referencing to more clearly illustrate which part of the code the information is about. Finally, if the programmer clicks on the indicator, this will open the default active view, the ribbon display (D) with buttons for reading and working with the versions of that artifact, as described next.

B. Navigating Versions

The “ribbon display” shown in Figure 1D is the default way Verdant shows all versions for an artifact, lined up side by side to the right of the original artifact. Unlike existing code interactions like a linter or autocomplete, where a pop-up may appear in the active text to supply short static information, versioning data is comprised of a long ordered list of information and must continuously update as the data scientist runs their code. So in the ribbon visualization, because code

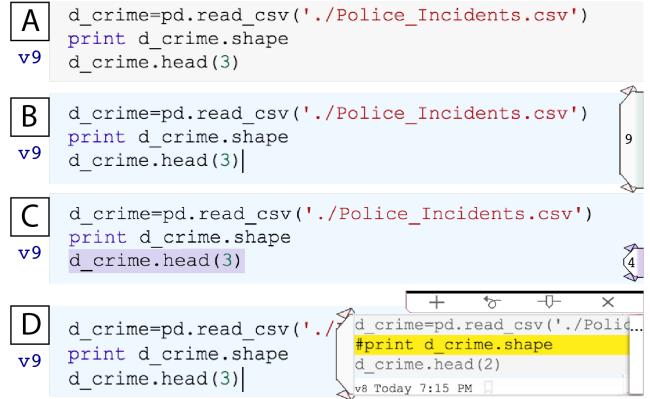


Figure 2. (A) no version information shown, (B) on selecting a cell an margin indicator displays how many versions of that artifact there are, (C) on selecting specific code the indicator updates to show how many versions of that specific snippet there are, (D) upon clicking the indicator, a ribbon visualization shows versions of an artifact starting with the most recent

and cells in the notebook are read from top to bottom, the version property of an artifact is visualized left to right, with the leftmost version, which is shown in blue (Figure 2D), always being the *active version*. Here “active version” will always refer to the version of the artifact that is in the notebook interface itself and that is run when the user hits the run button in the notebook. Since the ribbon is a horizontal display, it can be navigated by horizontal scroll, the right and left arrow keys, or by clicking the ellipsis bar at the far right of the ribbon which will open a drop-down menu of all versions (Figure 2D). For navigating versions, note that Variolite made a different design decision, and had users switch between versions via tabs. However, as suggested by Variolite’s usability study [4], as well as recent work on tab interfaces in general [30], tab interfaces do not scale well as the number of versions increases beyond a handful. On the other hand, choosing from a list display is not the most speedy for

retrieving an often used version if it is far down on the list. The ribbon display always shows the most *recent* versions first, making recent work fast to retrieve on the intuition that recent work is more likely to be relevant to the user's current task. For non-recent items, bookmarking is a standard interaction for fast retrieval of often-used items. Since robust history tools currently do not exist, we lack grounded data on which history versions a data scientist is likely to use. So to probe this question with real data scientists, we added an inactive bookmark icon to all versions, indicating that the user *can* bookmark it. We use this during our initial usability test to probe potential users on bookmarking, their use cases (if any) for it, and on various ways it could be displayed (see below).

C. Comparing Versions

Among many versions, it is important for a data scientist to quickly pick out what is important about that version out of lots of redundant content. This also helps provide "scents" for users to further forage for information. If a data scientist Lucy opens a cell's version and sees that only a certain line has changed much over the past month, she can adjust the ribbon to show only versions in which that line changed, hiding all other versions that are not relevant to that change.

In Verdant, a diff is shown in the ribbon and timeline views by highlighting different parts of a prior version in bright yellow (Figure 2D). For code, Verdant runs a textual diff algorithm consistent with Git, and for artifacts like tables that are rendered through HTML, Verdant runs a textual diff on the HTML versions and then highlights the differing HTML elements.

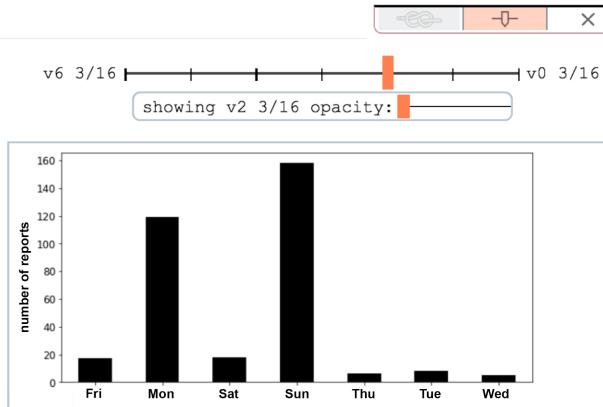


Figure 3. Timeline view. By dragging the top orange bar side to side the user can change the version shown. By dragging the lower orange bar, the user can set the opacity of the historical version they are viewing, in order to see it overlaid on top of the currently active output version.

Since an "artifact" can be a tiny code snippet or a gigantic table or a graph or a large chunk of code, one-size-fits-all is not the best strategy for navigation and visualization across all these different types. For instance, for visual artifacts like tables or images, visualization research [31] has found that side-by-side displays can make it difficult to "spot the difference" between two versions. In the menu bar that

appears with the default ribbon, a user can select a different way of viewing their versions. For visual artifacts, overlaying two versions is suggested, so a timeline view can be activated (Figure 3), by selecting the ∇ symbol. A data scientist can navigate the timeline view by dragging along the timeline, or by using the right/left arrow keys.

For visual diffing, Verdant again relies on advice from visualization research [31] and uses opacity in the timeline view where the user can change the opacity of a version they are looking at to see it overlaid on top of their currently active version.

For all artifact types, there are multiple kinds of comparisons that could be made, each of which optimizes for a different (reasonably possible) task goal:

1. What is different between the active version and a given prior version?
2. What changed in version N from the version immediately prior?
3. What changed in version N from version M, where M and N are versions selected by the data scientist from a list of versions?

For an initial prototype of Verdant, we chose to implement the first option only, on the hypothesis that spotting the difference between the data scientist's immediate current task and any given version will be most useful for spotting useful versions of their current task out of a list. We then used usability testing to probe through discussion with data scientists which kinds of diff they expect to see, and what task needs for diffs they find important (see study below).

D. Searching & Navigating a Notebook's Full Past

In-line versioning interactions allow users to quickly retrieve versions of artifacts present in their immediate working notebook, but has the drawback that some versions cannot be retrieved this way. The cell structure of a notebook evolves as a data scientist iterates on their ideas and adds, recombines, and deletes cells as they work [5]. Suppose that Lucy once had a cell in the notebook to plot a certain graph, but later deleted it once that cell was no longer needed. To recover versions of the graph, Lucy cannot use the in-line versioning discussed above, because that artifact no longer exists whatsoever in the notebook: she has no cell to point to and indicate to "show me versions of this". So to navigate to versions not in the present workspace, and to perform searches, Verdant also represents all versions in a list side pane.

The list pane can be opened by the user with a button, and is tightly coupled with the other visualizations such that if the user selects an artifact in the notebook, the pane will update to list all versions of that artifact, and stay consistent with the current selected version. If no artifact is selected, the list shows all versions of the notebook itself. With a view of the entire notebook's history, the user can see a chronologically ordered change list beginning with the most recent changes

across all cells in the notebook. Say Lucy wants to retrieve a result she produced last Wednesday that has since been deleted from her current notebook. Either by scrolling down the list or by using the search bar to filter the list by date, she can navigate to versions of her notebook from last Wednesday to try to pull out the relevant artifact when it last existed. Alternatively, she can use the search bar to look for the result by name. Note that Lucy does not need to actually find the *exact* version she is looking for from this list. Using foraging, if she can find the old cell in the list that she thinks at some point produced the result she is thinking about, she can select that cell in the list to pull up all of its versions of code and output. From there, she can narrow her view further to only show the output produced on Wednesday. This method of searching relies on following clues across dates and dependency links among artifact versions, rather than requiring the data scientist to recall precise information that would be needed for a query in a language like Prolog [13].

```

 v12 3/17 2:30 PM
d_crime=pd.read_csv('./Police_Incidents.csv')
print d_crime.shape
d_crime.head(21)

 v11 3/16 5:17 PM
d_crime=pd.read_csv('./Police_Incidents.csv')
print d_crime.shape
d_crime.head(21)

 v10 3/16 3:42 PM
d_crime=pd.read_csv('./Police_Incidents.csv')
print d_crime.shape
d_crime.head(23)

 v9 3/16 3:42 PM

```

Figure 4. In the list view, the user can select one or more versions to act upon. With the search bar, the user can filter versions using keywords or dates.

VI. REMIX, REUSE, & REPRODUCE

When data scientists produce a series of results, they may later be required to recheck how that result was produced. Common scenarios include inspecting the code that was used to check that the result is trustworthy, or reproducing the same analysis on new data [5]. Without history, reproducing results is commonly a tedious manual process, where the data scientist re-creates the original code from memory [5].

A. Replay older versions

To replay any older version of an artifact in the notebook, a user in Verdant can make that version the *active* version and then re-run their code. In any of the in-line or list visualizations of an artifact's versions, the data scientist can select an older version of an artifact and use the  symbol button to make that version the active one. The formerly active version for that artifact is not lost, since it is recorded and added as the most recent version in the version list. If a data scientist wants to replay a version of an artifact that no longer exists in the current notebook, that artifact will be added as a new cell of the current notebook, located as close as possible to where it was originally positioned.

Although this interaction can be used to make any older version the active one, it completely ignores dependencies that the older version originally had. Our rationale behind this is clarity and transparency: if Lucy clicks the  symbol on a certain version, that changes only the artifact the version belongs to. If instead Verdant also updated the rest of the notebook, changing other parts of the notebook to be consistent with the version dependencies, then Lucy may have no understanding of what has changed. In addition, sometimes data scientists use versions more as a few different options for doing a particular thing (e.g., to try a few different ways for computing text-similarity) and are not interested in the last context the code-snippet-version was run in, just in reusing the specific selected code-snippet-version. To work with prior experiment dependencies, Verdant provides a feature called “Recipes”, described next.

B. Output Recipes

What code should a data scientist re-run to reproduce a certain output? Once the data scientist finds the output they would like to reproduce, they can use the  symbol button (shown at the top of Figure 3). Verdant uses the chain of dependency links that it has calculated from the output to produce a *recipe visualization*, shown in Figure 5. The “recipe” appears in the side list pane as an ordered list of versions labeled “step 1” to “step N”. Consistent with all other visualizations, the recipe highlights in yellow any code in the steps that is different from the currently showing code in the notebook. So, if a code cell is entirely absent from the notebook, it will be shown in entirely yellow. If a matching code cell already exists in the notebook and perfectly matches the active version, it will be shown in entirely grey in the recipe with a link to navigate to the existing cell to indicate that the data scientist can just run the currently active version of that code. Note this dependency information is imperfect, because we do not version the underlying data files used, so if the dataset itself has changed, the newly produced output may be still different than the old one. We discuss several avenues for versioning these data structures in Future Work.

The screenshot shows a software application window titled 'Verdant'. At the top, there is a search bar and a button labeled 'to reproduce output:'. Below this is a table with columns 'IncidentNum', 'Category', 'Descript', and 'Date'. Two rows are visible: one for 'NON-CRIMINAL' LOST PROPERTY and another for 'ROBBERY' ROBBERY, BODILY FORCE. Below the table are two code cells. The first cell, labeled 'step # 0 v0 Today 7:15 PM', contains Python code for importing libraries like sys, os, pandas, numpy, matplotlib, and seaborn, and for running %matplotlib inline. The second cell, labeled 'step # 1 v9 Today 7:15 PM', contains code to read a CSV file named 'Police_Incidents.csv' and print its shape.

Figure 5. In the recipe view, a user sees the output they selected first, and then an ordered series of code cells that need to be run to recreate that output.

B. Trust and Relevancy of versions at scale

A data scientist will try many attempts during their experimentation, many of which may be less successful or flawed paths [4], [32]. Thus, especially when collaborating with others, it can be important for a data scientist to communicate which paths failed [5], [8] and how they got to a certain solution. “Which path failed” requires a kind of storytelling that it unlikely automated methods can capture, thus it would be most accurate for the data scientist to label certain key versions themselves. However, we know from how software engineers use commits (often lacking clear organization or naming) that programmers can be reluctant to spend any time on organizing or meaningfully labeling their history data [8]. Under what circumstances would data scientists be motivated to label trustworthiness and relevancy of their code? To experiment with interactions for this purpose, Verdant uses an interaction metaphor of email in the version list. Like in email, the data scientist can select one or many of their versions from the list (filtering by date or other properties through the search bar) and can “archive” these versions so that they are not shown by default in the version list. Also, the data scientist can mark the versions as “buggy” to more strictly hide the versions and label them as artifacts that contain dangerous or poor code that should not be used. If an item is archived or marked buggy, it still exists in the full list view of versions (so that it can be reopened at any time), but it is hidden by being collapsed. If an item is archived or marked buggy and has direct output, those outputs will be automatically archived or marked buggy as well. A benefit of using a familiar metaphor like archiving email for a prototype system is that it is much easier to communicate the intent of this feature to users. During the usability study, discussed below, we showed the archive and “buggy” buttons to data

scientists to probe how, if, and under what tasks they would actively manually tag versions like this.

VII. INITIAL USABILITY TEST

Verdant is a prototype that introduces multiple novel types of history interaction in a computational notebook editor. Thus it is necessary to test both the usability of these interactions and also to investigate through interview probes how well these interactions seem to or meet real use cases to validate that our designs are on the right track.

For our study setup, we aimed to create semi-realistic data analysis tasks and history data. For Verdant to store and show data science history at scale and in realistic use, we anticipate a later stage field study where data scientists would work on their own analysis tasks in the tool over days and weeks. Here for an initial study, we avoid participants having to work extensively on creating analysis code by instead asking them to use Verdant to try to *navigate* and *comprehend* the history of a fictitious collaborator’s notebook. To create realism, we chose this notebook out of an online repository of community-created Jupyter notebooks that are curated for quality by the Jupyter project [33]. From this repository we searched for notebooks that contained very simple exploratory analyses and that needed no domain-specific knowledge to ensure the notebook content would not be a learning barrier to participants. The notebook we chose does basic visualizations of police report data from San Francisco [34]. Since currently detailed history data is not available for notebooks, we edited and ran different variations of the San Francisco notebook code ourselves to generate a semi-realistic exploration history.

Next, we recruited individuals who A) had data science programming experience, B) were familiar with Python, and C) had at least two months experience working with Jupyter notebooks. This resulted in five graduate student participants (1 female, 4 male) with an average of 12 years of programming experience, an average of 6 years of experience working with data, and an average of 3 years experience using notebooks. In a series of small tasks, participants were asked to navigate to different versions of different code, table, and plot artifacts using the ribbon visualization, diffs, and timeline visualization. The study lasted from 30 to 50 minutes and participants were compensated \$20 for their time. Participants will be referred to as P1 to P5.

All participants were able to successfully complete the tasks, suggesting at least a basic level of usability. Among even a small sample, we were surprised by the diverse use cases participants expressed that they had for the tool. P1 and P5 expressed that they would like to use the ribbon visualization of their versions about every 1-2 days to reflect on their experiment’s progress or backtrack to a prior version. P2 was largely uninterested in viewing version history, but instead was enthusiastic about using the ribbon visualization to switch between 2 to 4 different variants of an idea. P3 was less interested in viewing version history of code cells, but greatly

valued the ability to view and compare the version history of output cells. P3 commonly ran models that took a long time to compute (so they only wanted to run a certain version once), and currently to compare visual outputs, had to scroll up and down their notebook. However, P3 did appreciate the ability to version a code cell, as a safe way of keeping their former work in case they wanted to backtrack later. Finally P4 primarily used notebooks in their classwork, and were very enthusiastic about using code artifact history to debug, revert to prior versions, and to communicate to a teaching assistant what methods they had attempted so far when they went to ask for help. P2, P3, and P4 expressed they would like to use the inline history visualizations “all the time” when doing a specific kind of task they were interested in, whether that be comparing outputs or code.

In this initial study, a participant’s imagined use case affected which features of Verdant they cared about most. When probed about the use of bookmarking, P2 felt strongly that bookmarks would be useful for their use case of switching among a few different alternative versions, however the other participants who had a more history-based use case were neutral about bookmarking. For the probe in which we showed email-like buttons for archiving or marking code as buggy, participants had very divergent opinions. P1 said they would want to mark versions as buggy and said that they would want to group a bunch of versions and leave a note about what the problem was, but would never use the archive functionality. P2 said they would likely mark versions as buggy, but would be wary of using the archive button to hide older or unsuccessful content. P2 disliked the “archive” metaphor because they felt the relevance of different versions was too task dependent: a version that seems worth archiving in one task context might be very relevant for a future task. All other participants were neutral about the two options, and saw themselves using them to curate their work occasionally. While participants said they would use the inline visualizations daily or every other day when working within a notebook, they said they would use the list pane or recipe visualizations only once a week or once a month. P5 said that although they imagined themselves tracing an output’s dependency rarely, this feature was extremely valuable to them when needed, since currently when P5 must recreate output, this was a tedious and error-prone manual process of trying to re-code its dependencies from memory.

In terms of diffing, all five participants were familiar with and used Git, and all guessed that the yellow-highlighted diff in Verdant, like Git, showed what had changed from one version to the next. When we clarified that yellow highlighting showed the diff between any version and the *active* version of the artifact, two participants said that was actually more helpful for them to pick which other versions to work with. All participants wanted the option of multiple kinds of diff. P3, who primarily wanted to diff output, asked for more kinds of visual diffing than the timeline scroll such as setting opacity

to see two versions overlayed (which we added into the current Verdant) and a yellow-highlighting for image diffs. Finally, multiple participants disliked horizontal scroll for navigating the ribbon visualization (horizontal scroll is not a gesture on many mice devices) and preferred the ribbon’s dropdown menu to select versions.

VIII. FUTURE WORK & CONCLUSION

There remain many key directions in supporting experiment history. Our small user study revealed a high variance of an individual’s day-to-day task needs for their history. Thus to truly understand the impact of Verdant and future tools in this space, a key next step is to conduct a field study across a larger number of participants over several weeks in order to collect grounded data on how data scientists put history to use in practice. There remain many further systems design directions as well, particularly to visualize differences and comparisons between different kinds of artifacts. While we demonstrate Verdant on images and code, different visualizations may be more useful and effective to portray the history of tables, plots, or notes. Future work is also needed to help collect version information about data files, to ensure reproducibility without consuming too much memory space. In work such as Verdant, we move from considering code history only for engineering practice to building human-centered history tools for experts and end-user programmers to synthesize their ideas, and more responsibly conduct experimentation and exploration.

ACKNOWLEDGMENTS

We thank our pilot participants. This research was supported in part by a grant from Bloomberg L.P., and in part by NSF grant IIS-1314356. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the funders.

REFERENCES

- [1] D. Fisher, R. DeLine, M. Czerwinski, and S. Drucker, “Interactions with big data analytics,” *Interactions*, vol. 19, no. 3, pp. 50–59, 2012.
- [2] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer, “Enterprise Data Analysis and Visualization: An Interview Study,” *IEEE Trans. Vis. Comput. Graph.*, vol. 18, no. 12, pp. 2917–2926, Dec. 2012.
- [3] P. J. Guo, “Software tools to facilitate research programming,” Doctor of Philosophy, Stanford University, 2012.
- [4] M. B. Kery, A. Horvath, and B. A. Myers, “Variolite: Supporting Exploratory Programming by Data Scientists,” in *ACM CHI Conference on Human Factors in Computing Systems*, 2017, pp. 1265–1276.
- [5] M. B. Kery, M. Radensky, M. Arya, B. E. John, and B. A. Myers, “The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool,” in *ACM CHI Conference on Human Factors in Computing*

- [6] Systems, 2018, p. 174.
- [7] A. Rule, A. Tabard, and J. Hollan, “Exploration and Explanation in Computational Notebooks,” in *ACM CHI Conference on Human Factors in Computing Systems*, 2018, p. 32.
- [8] “Jupyter Notebook 2015 UX Survey Results,” *Jupyter Project Github Repository*, 12/2015. [Online]. Available: https://github.com/jupyter/surveys/blob/master/surveys/2015-12-notebook-ux/analysis/report_dashboard.ipynb.
- [9] M. Codoban, S. S. Ragavan, D. Dig, and B. Bailey, “Software history under the lens: a study on why and how developers examine it,” in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, 2015, pp. 1–10.
- [10] K. Patel, “Lowering the barrier to applying machine learning,” in *Adjunct proceedings of the 23nd annual ACM symposium on User interface software and technology*, 2010, pp. 355–358.
- [11] A. Tabard, W. E. Mackay, and E. Eastmond, “From Individual to Collaborative: The Evolution of Prism, a Hybrid Laboratory Notebook,” in *Proceedings of the 2008 ACM Conference on Computer Supported Cooperative Work*, San Diego, CA, USA, 2008, pp. 569–578.
- [12] F. Pérez and B. E. Granger, “IPython: a System for Interactive Scientific Computing,” *Computing in Science and Engineering*, vol. 9, no. 3, pp. 21–29, May 2007.
- [13] “provenance - Wiktionary.” [Online]. Available: <https://en.wiktionary.org/wiki/provenance>. [Accessed: 22-Apr-2018].
- [14] J. F. N. Pimentel, V. Braganholo, L. Murta, and J. Freire, “Collecting and analyzing provenance on interactive notebooks: when IPython meets noWorkflow,” in *Workshop on the Theory and Practice of Provenance (TaPP), Edinburgh, Scotland*, 2015, pp. 155–167.
- [15] P. J. Guo and M. I. Seltzer, “Burrito: Wrapping your lab notebook in computational infrastructure,” in *4th USENIX Workshop on Theory and Practice of Provenance*, 2012.
- [16] S. Srinivasa Ragavan, S. K. Kuttal, C. Hill, A. Sarma, D. Piorkowski, and M. Burnett, “Foraging Among an Overabundance of Similar Variants,” in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, San Jose, California, USA, 2016, pp. 3509–3521.
- [17] “verdant - Wiktionary.” [Online]. Available: <https://en.wiktionary.org/wiki/verdant>. [Accessed: 22-Apr-2018].
- [18] R. J. Brunner and E. J. Kim, “Teaching Data Science,” *Procedia Comput. Sci.*, vol. 80, pp. 1947–1956, Jan. 2016.
- [19] D. E. Knuth, “Literate programming,” *Comput. J.*, vol. 27, no. 2, pp. 97–111, 1984.
- [20] “Colaboratory,” 2018. [Online]. Available: <https://colab.research.google.com/>. [Accessed: 22-Apr-2018].
- [21] K. Cheung and J. Hunter, “Provenance explorer--customized provenance views using semantic inferencing,” in *International Semantic Web Conference*, 2006, pp. 215–227.
- [22] I. Herman, G. Melançon, and M. S. Marshall, “Graph visualization and navigation in information visualization: A survey,” *IEEE Trans. Vis. Comput. Graph.*, vol. 6, no. 1, pp. 24–43, 2000.
- [23] S. Chacon and B. Straub, “Git and Other Systems,” in *Pro Git*, S. Chacon and B. Straub, Eds. Berkeley, CA: Apress, 2014, pp. 307–356.
- [24] R. Team and Others, “RStudio: integrated development for R,” *RStudio, Inc. , Boston, MA URL http://www.rstudio.com*, 2015.
- [25] Y. Yoon, B. A. Myers, and S. Koo, “Visualization of fine-grained code change history,” in *2013 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2013, pp. 119–126.
- [26] I. Neamtiu, J. S. Foster, and M. Hicks, “Understanding source code evolution using abstract syntax tree matching,” *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005.
- [27] R. Koschke, R. Falke, and P. Frenzel, “Clone Detection Using Abstract Syntax Suffix Trees,” in *2006 13th Working Conference on Reverse Engineering*, 2006.
- [28] S. Oney and B. Myers, “FireCrystal: Understanding interactive behaviors in dynamic web pages,” in *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2009, pp. 105–108.
- [29] “IntelliJ IDEA,” *IntelliJ IDEA*. [Online]. Available: <https://www.jetbrains.com/idea/>. [Accessed: Apr-2017].
- [30] Nathan Hahn, Joseph Chee Chang, Aniket Kittur, “Bento Browser: Complex Mobile Search Without Tabs,” in *2018 CHI Conference on Human Factors in Computing Systems*, 2018, p. 251.
- [31] M. Gleicher, D. Albers, R. Walker, I. Jusufi, C. D. Hansen, and J. C. Roberts, “Visual comparison for information visualization,” *Information Visualization; Thousand Oaks*, vol. 10, no. 4, pp. 289–309, Oct. 2011.
- [32] K. Patel, J. Fogarty, J. A. Landay, and B. Harrison, “Investigating statistical machine learning as a tool for software development,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2008, pp. 667–676.
- [33] “A gallery of interesting Jupyter Notebooks.” [Online]. Available: <https://github.com/jupyter/jupyter/wiki/A-gallery-of-interesting-Jupyter-Notebooks>. [Accessed: 24-Apr-2018].
- [34] lmart, “SF GIS CRIME,” *GitHub*. [Online]. Available: <https://github.com/lmart999/GIS>. [Accessed: 27-Apr-2018]

Supporting Remote Real-Time Expert Help: Opportunities and Challenges for Novice 3D Modelers

Parmit K. Chilana¹, Nathaniel Hudson², Srinjita Bhaduri³, Prashant Shashikumar⁴ and Shaun Kane⁵
^{1,4} Simon Fraser University ² Ross Video ^{3,5} University of Colorado-Boulder
 Burnaby, BC, Canada Ottawa, ON, Canada Boulder, CO, USA
 {pchilana, pshashik}@sfu.ca me@nathanielh.com {srinjita.bhaduri, shaun.kane}@colorado.edu

Abstract—We investigate how novice 3D modelers can remotely leverage real-time expert help to aid their learning tasks. We first carried out an observational study of remote novice-expert pairs of 3D modelers to understand traditional chat-based assistance in the context of learning 3D modeling. Next, we designed *MarmalAid*, a web-based 3D modeling tool with a novel real-time, in-context help feature that allows users to embed real-time chat conversations at any location within the 3D geometry of their models. Our user study with 12 novices who used both *MarmalAid*'s real-time, in-context chat and an external chat tool to seek help, showed that novices found the real-time, in-context chat to be more useful and easier to use, and that experts asked for fewer clarifications, allowing the novices to ask more task-related questions. Our findings suggest several design opportunities to utilize and extend the real-time, in-context help concept in 3D modeling applications and beyond.

Keywords—real-time help; in-context help; software learnability; 3D modeling

I. INTRODUCTION

The proliferation of 3D printing, virtual reality (VR), and augmented reality (AR) applications has sparked wide interest in learning 3D modeling. Although beginner-friendly modeling tools, instructional materials, and training programs are increasingly available, learning 3D modeling can still be daunting [17,21]. Prior work has documented many usability and learnability problems in using complex 3D modeling software, including dealing with confusing terminologies, creating complex shapes, and interacting with unfamiliar 3D geometry [4,27,37].

To develop 3D modeling skills, some newcomers seek help directly from experts, rather than learning from static videos or tutorials [21]. A novice 3D modeler may attend workshops at the local library, where they can receive one-on-one assistance from a workshop instructor throughout the modeling process [21]. Other learners may join maker spaces and online communities to learn modeling techniques from their peers [30,36,42]. By working with experts, modelers can experience “over-the-shoulder-learning” [39] and can ask targeted questions that reflect their particular software, task, and 3D model.

Despite the benefits of one-on-one help, it is rarely available outside of formal learning environments. Although novices can seek help from remote experts by posting a question on a discussion forum or asking a friend over email, it can be difficult for the expert to provide help without direct access to the user’s task, and without the ability to quickly ask follow-up questions [8]. Furthermore, when requesting asynchronous remote help online, hours or even days may pass before a response arrives [3].

In this paper, we investigate how remote, real-time expert can be designed for 3D modeling tasks and how this form of help is used and perceived by novice 3D modelers. We carried out an observational study of 6 novice-expert pairs where each novice built a 3D model while asking questions to a remote expert via web chat. We found that although the novice users benefited by directly interacting with an expert, they had trouble explaining the visual context of their modeling tasks, often prompting clarification questions from experts.

To support for remote, real-time communication with experts, and drawing inspiration from modern contextual help approaches [9,16,20,29], we designed *MarmalAid*, a web-based 3D modeling application with an embedded *real-time, in-context chat feature* (Fig. 1). *MarmalAid* allows users to easily share work-in-progress with experts by establishing a shared visual context, and enables users to start real-time, in-context conversations that target specific areas of their 3D model. To evaluate *MarmalAid*, we recruited another set of 12 novice 3D modelers to compare *MarmalAid*'s real-time, in-context chat feature with an external chat application. Our results show that the majority of the participants found the real-time, in-context chat to be more useful and easier to use than the external chat. Experts asked for fewer clarifications when using the real-time, in-context chat feature, enabling the novices to ask more task-related questions. Participants' qualitative feedback confirmed their strong preference for *MarmalAid*'s real-time, in-context chat.

The main contributions of this paper are: 1) the design of a real-time, in-context chat feature for enabling remote one-on-one synchronous communication between novice and expert 3D modelers; and 2) empirical insights into how novice 3D modelers use and perceive remote expert help via real-time, in-context chat vs. an external chat application and how it affects their 3D learning tasks. Although *MarmalAid*'s help features were designed to support 3D modeling tasks, we reflect on how the lessons learned from this work can generalize to other creative design tasks and how the real-time, in-context help approach can augment other forms of software learning and troubleshooting.

II. RELATED WORK

A. Learning by Demonstration

There is a long history of HCI research exploring learning to use feature-rich software by demonstration. Some approaches have investigated the use of animated steps (e.g., [19, 31]), interactive, guided tutorials (e.g., [12, 15, 23]), and even complete application workflows and editing histories (e.g., [18]). Other work has explored video-based demonstrations and mixed multimedia forms of assistance (e.g., [7,16,33]). These approaches

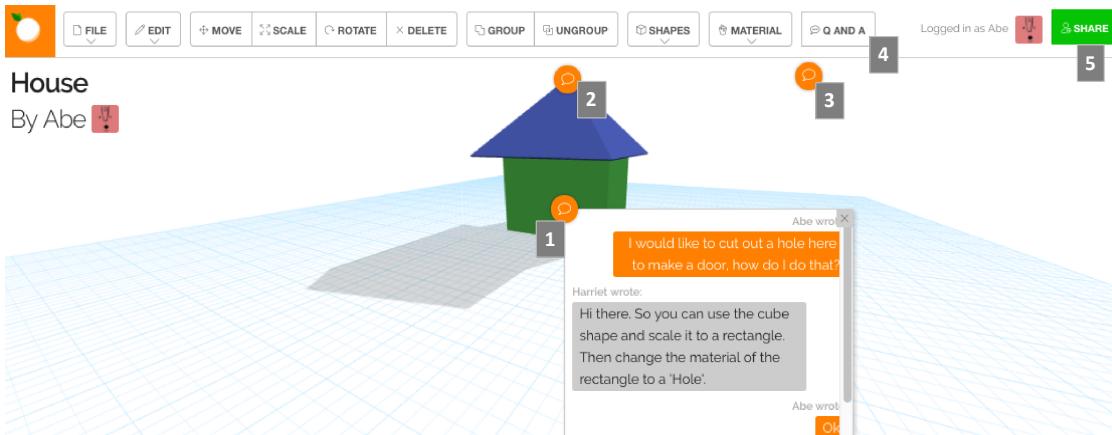


Fig. 1. MarmalAid user interface: (1) users can anchor chat windows to request remote assistance on any part of their model; (2) by clicking on the comment icon, the chat window collapses but remains attached; (3) document-level chat windows can also be added without a model-specific anchor; (4) the Q and A icon is used to create a new chat window; (5) the Share feature can be used to obtain a model-specific URL for sharing.

enable novices to follow steps created by expert users or application designers rather than scouring through static help materials. Despite the benefits of learning by demonstration, novices face a number of challenges in following the steps of experts [41], often giving up and seeking one-on-one help [21].

Recent research has explored how to enhance instructional materials with community contributions and embedded Q&A discussions (e.g., [5,24,40]). Our key motivation for MarmalAid was to investigate how one-on-one synchronous help can be offered within the application, in context of the users' tasks. We see in-context real-time chat features as complementing other forms of learning by demonstration and community-based help.

B. In-Application Contextual Help

HCI researchers have also long explored methods for providing contextual help, such as by attaching help messages to specific UI labels and controls within an application. Some of the prominent contextual help approaches include text and video-based tooltips [16], *Balloon Help* [11], pressing F1 over UI elements, choosing a command name to see animated steps [38], among others. These approaches usually require the help content to be pre-authored at design time, making the content limited to explaining the functionality of a specific UI widget or feature. In response to this limitation, recent work has explored the idea of crowdsourced contextual help, with systems such as *Lemon-Aid* [9], *IP-QAT* [29] and *HelpMeOut* [20] where the in-context help can be dynamically created and maintained by the user community based on their actual application needs.

Although MarmalAid is inspired by these existing forms of contextual help, it offers two fundamental differences: 1) there is no pre-authored help content or ability to see previous community-authored questions—instead, MarmalAid provides help that is customized for a user's specific task and questions; and, 2) MarmalAid's real-time help can be directly anchored to part of a 3D model's geometry, raising the possibility of attaching Q&A to a broader set of anchors and referents within an application. As we will discuss in our findings, study participants asked proportionally more domain-specific questions (and fewer UI questions) when using Marmalaid's in-context help.

C. Synchronous Remote Help

Another area of HCI research that relates to the present work is the exploration of synchronous remote help-giving in learning and troubleshooting contexts. For example, Crabtree et al. [10] compared help given at a library help desk to help given at a call center of a printer manufacturing company, and found that users lacked precise technical knowledge of the machine that they used and had limited understanding of how to move from their current state to a solution. Similarly, studies of remote troubleshooting in the domain of home networking [34,35] have shown that users seeking help via telephone often struggle in providing precise descriptions of their issues, making it difficult for support specialists to understand and diagnose their problems.

Given the difficulty of diagnosing user-reported issues remotely, even when assistance can be provided in real-time, some have argued that shared visual context is necessary to facilitate conversational grounding during Q&A [14]. MarmalAid builds on this idea by providing a shared visual space for Q&A and by enabling users to spatially anchor questions to specific points of interest. Our empirical findings complement prior studies of remote help-giving and provide insights into how real-time remote help may be used during collaborative 3D modeling.

III. OBSERVATIONAL STUDY OF NOVICE-EXPERT PAIRS

The goal of our initial observational study was to better understand how novice users interact with a remote expert when learning how to create 3D models, and to identify the challenges faced by both novices and experts during this remote activity. Novices created 3D models using Autodesk *Tinkercad*, a web-based 3D modeling application, and talked with the expert via the *Slack* team chat application.

We recruited 10 participants from the local community via email to student lists on campus and posts to nearby makerspaces. Participants were all between the ages of 21 to 56. Based on self-report, we classified six participants (2 female) as novices, and four participants (1 female) as experts. Novice participants had little to no prior experience in 3D modeling, while the experts were familiar and/or proficient in using several CAD tools, such as *SketchUp*, *Blender*, *Rhino*, *Maya*, and *Solidworks*. Participant occupations included student and factory worker.

Each study session included one novice participant and one expert participant. Each novice only participated in one study

session; two experts participated in one session only, and two experts participated in two study sessions. After an initial introduction, both novice and expert were placed in separate rooms. Each participant was seated at a computer that was running the Tinkercad and Slack applications. Expert participants were instructed to answer the novice's questions via Slack's text chat. Study sessions took place at a university research lab. Each session lasted one hour. Participants were compensated \$15.

A. Procedure

Each novice participant completed two 3D modeling activities: an initial tutorial that introduced Tinkercad, followed by a self-guided 3D modeling task.

At the start of the study session, the facilitator introduced the novice participant to the goal of the study session. The novice then watched a 3-minute video tutorial that demonstrated the basics of Tinkercad, and then tried to reproduce the models shown in the tutorial using Tinkercad for a total of 10 minutes. Expert participants were given the option of completing the Tinkercad tutorial but were not required to do so.

After completing the tutorial, novice participants were shown an image of the 3D model and were instructed to recreate that model in Tinkercad. The test model was a set of 3D letter shapes; this model was designed so that novice participants would need to use all of Tinkercad's basic features, including placing 3D primitives (e.g., box, cylinder), changing the size and orientation of primitives, and combining primitives to produce more complex shapes. Novices were given 30 minutes to complete as much of the 3D modeling task as possible and were instructed to ask the expert for help via Slack chat. During this activity, they were asked to think-aloud so that the research team member could better observe what the novice user was doing.

Each novice and expert participant filled out a demographic questionnaire before beginning the study. At the end of the study, both participants were interviewed by the facilitator, and both participants filled out feedback questionnaires.

We captured participants' computer screens, and recorded audio of the think-aloud activity and follow-up interview. We also collected the 3D models created in Tinkercad and logged the Slack conversations between pairs of participants. Analysis of the collected data was done collaboratively by the research team using affinity diagrams and inductive analysis. Together, we identified the challenges that novice participants experienced in creating 3D models, observed patterns of help-seeking behavior between novices and experts, and analyzed feedback from participants about this type of collaborative learning activity.

B. Key Findings and Implications

Our analysis of observations, chat logs, and subjective feedback suggested that novice participants were positive about the idea of working remotely with an expert as they learned how to create 3D models but encountered some key challenges.

Challenges in Formulating Questions: The chat logs from the study showed that novices experienced difficulties in framing their questions, and in describing the current state of their 3D model. When novices encountered a problem in creating their model or manipulating Tinkercad's view, they often lacked

the language to describe their problem. As a result, the expert participant struggled in understanding what the novice was asking and ended up asking multiple follow-up questions to understand the novice's help needs.

For example, one participant wished to move Tinkercad's camera to a specific location, but had difficulty describing exactly how they wished to position the camera, writing:

...the default camera view centers the camera over the center of the workplane. I'd like to be able to look directly down from the top left corner of the workplane so I can get a better idea of how my objects are positioned... (U04)

Expert participants asked follow-up questions in all six study sessions. In the above example, the expert asked several follow-up questions to clarify the novice's intent. In another example, the expert participant did not understand what the novice wished to do; asking a follow-up question resulted in the novice providing a clearer description of the goal:

U08: How do I change the angle of a shape?

Expert: What exactly do you mean?

U08: To rotate in the 3D plane rather than just 90° up or down

Post-activity interviews confirmed that both novices and experts struggled to communicate about specific problems. Five of the six novices explicitly mentioned the difficulty in articulating questions. One participant noted that she was aware that 3D modeling had specific terminology, but had difficulty articulating questions because she did not know that terminology:

...[Tinkercad] had some explicit terminology that made sense to me but was not immediately transferred to the expert...so, there were few rounds of back and forth clarifications... (U02)

The novices' difficulties in articulating questions, and experts in answering questions, suggests that there is an opportunity to improve remote help by providing more shared points of reference between the novice and expert.

Challenges in Conveying the Visual Context: In traditional, collocated help sessions, the expert can directly observe the novice's work; this shared visual context can help the expert to understand the novice's goals and can enable the expert to provide more contextually relevant feedback. For our observational study, we focused on understanding how the novice and expert participants working remotely might create workarounds to compensate for the lack of shared visual context.

Novice participants were shown how to capture screenshots and share them via Slack. Experts sometimes asked for a screenshot to help them understand what the novice was asking about. The exchange below shows how a participant had difficulty finding a menu item to change color and how the expert asked for a screenshot of the novice's view:

Expert: if you click on the shape, the dropdown menu has a color select menu

U06: For some reason, I don't see it

Expert: hmmmm can you send me a screenshot of the window?

Expert: ...[After seeing the screenshot] you can click on the red circle above "solid" to open the color menu

In some cases, novice participants sent screenshots even without any prompting from the expert, such as when asking about Tinkercad features that they did not know the names of or when they did not know how to describe specific 3D modeling

features like addition or subtraction of shapes. Overall, four of the six participants used screen-shots.

While sharing screenshots enabled the participants to create a shared visual context, this feature was something of a work-around and did not naturally fit into these participants' workflow. Several novice participants noted that creating and sharing screenshots required additional effort and offered suggestions for improving the process of sharing visual context:

Something more automated could have been better...Or have a playback for the model novice is building and the expert can look at it. Or like have a split-screen where expert can see in real-time and give feedback. (U01)

Overall, our observations and interviews suggest that shared visual context is helpful in both asking for help and offering help. However, using external tools to create and share screenshots added some friction to the 3D modeling task. These findings suggest that enriching the shared visual context between novices and experts and integrating the Q&A process into the 3D modeling workflow, could provide many of the benefits of in-person help without requiring an in-person expert. This insight helped inform the design goals for MarmalAid, a new 3D modeling tool that explicitly supports real-time remote help.

IV. MARMALAIID: DESIGN OF REAL-TIME, IN-CONTEXT HELP

Based on findings from our observational study, we designed MarmalAid to explore the benefits and challenges of offering real-time, in-context help within a new 3D modeling application. The design of MarmalAid was driven by two key goals: providing real-time, back and forth discussion of specific 3D modeling problems, and creating a shared visual context between two remote users, such as a novice and an expert.

A. Creating 3D Models

MarmalAid is designed to enable novices to create simple 3D models. MarmalAid's modeling tools are based upon those offered by Tinkercad. MarmalAid features a basic constructive solid geometry (CSG) modeling system in which predefined shapes (e.g., cubes, spheres, cylinders,) can be transformed by rotation, scaling and translation, and can be combined via addition or subtraction to create more complex shapes. To add holes and negative space to models, primitives can be marked as a *hole*; when a hole is combined with other primitives, the shape of the hole primitive is subtracted from the compound object.

B. Sharing the Visual Context

As novice participants in our observational study found the process of sharing screenshots to be cumbersome, MarmalAid allows a user to share their view of the 3D model in real time with a remote expert. MarmalAid offers novice users two mechanisms for sharing their models with others. At any point, a novice user can click the "Share" button (Fig. 1.5) and enter an expert's email address. This action will generate an email to the expert that contains a link to the active MarmalAid document. MarmalAid users can also share their current project by simply copying the document URL and pasting it into an email, chat message, or any other communication tool.

The expert can open the shared link to see the novice user's work in progress and tagged areas of the 3D model in the MarmalAid online editor. The expert's view is a live version of the

novice's model: changes made by the novice will be shown to the expert in the shared view, and vice versa. The expert can further explore and edit the shared model and participate in the in-context chat conversations. These sharing features enable a remote expert to provide feedback about the novice participant's work, and about the current 3D model, without the need to install additional software or send 3D model files back and forth.

C. Adding Real-Time, In-Context Chat Conversations

MarmalAid's core help feature is its real-time, in-context help chat system. MarmalAid allows novice users to create real-time chat sessions with experts that are embedded directly into MarmalAid's user interface (UI), enabling a user to seek help without switching to a different work context.

Geometry-specific Conversations: Since our observational study showed that novice 3D modelers may have difficulty articulating questions because they lack the appropriate terminology, MarmalAid allows users to tag a specific part of a 3D model when asking a question. This feature allows the novice user to direct the expert's attention, even when the novice cannot clearly describe their problem (Fig. 1.1).

To activate the geometry-specific chat conversation, the user clicks on MarmalAid's "Q and A" button (Fig. 1.4), and then clicks any point in the 3D model to create a new chat window (e.g., Fig. 1.1). The chat window anchors to the particular point on the object, so that both users can see the chat in the same location. Chat windows are anchored to the 3D model and remain at the same location even if the camera is moved, or if the object itself is rotated, translated, or scaled. Users can minimize chat windows when not in use, and re-open them later (Fig. 1.2). Users can create multiple chat windows in the same document.

Document-level Conversations: If a user is not sure where to ask a question, they can still attach a chat window outside of the geometry of the 3D model, within the MarmalAid document window (Fig. 1.3). Users can discuss the whole model or ask general questions through a single chat window in the sidebar of the MarmalAid document window.

In-Application Notifications: To support collaborative editing, MarmalAid tracks all actions and chat messages made on the document and notifies all users when a comment has been added or the 3D model has been modified (via an in-app notification).

Note that the current version of MarmalAid assumes that the user knows who they want to share their model with and does not support matchmaking with new experts. Future versions of MarmalAid may provide increased support for sharing help requests on community-based sites, such as discussion forums.

V. IMPLEMENTATION OF MARMALAIID

MarmalAid is an entirely web-based application designed to run on any modern desktop web browser without any need to install software. MarmalAid has been tested in Safari 9.1.1, Chrome 55, Firefox 47.0.1, and Internet Explorer 11.0.

MarmalAid's server-side component is written using Python 3's Flask web framework and stores data in a SQLite database. This server-side component manages user accounts and 3D model files, controls permissions for viewing and editing models, and sends notifications. MarmalAid's server uses the

Socket.IO library for real-time communication between the server and clients.

MarmalAid's user interface and 3D modeling tools are implemented in HTML5 and JavaScript and run directly in the browser. MarmalAid's 3D viewport is implemented using the Three.js library and is rendered via WebGL, which allows for responsive hardware-accelerated 3D in the browser. Geometric manipulations of the 3D models, such as moving, scaling, and combining objects, is handled by the CSG.js (constructive solid geometry) library. MarmalAid's web client uses Socket.IO for real-time communication and Require.js to manage loading of the various component modules.

VI. USER STUDY

To evaluate MarmalAid's real-time, in-context help features, we conducted a user study in which we observed how novice 3D modelers work with a remote expert while learning 3D modeling. For this study, we created two variants of MarmalAid: one that included all of MarmalAid's integrated chat features (*In-Context Chat*), and another that included MarmalAid's shared view, but relied on an external chat application (*External Chat*).

The goals of this study were to assess the strengths and weaknesses of interacting with an expert while using real-time, in-context help; and, to investigate whether help-seeking strategies differ when help is presented in-context vs. in externally.

We recruited 12 participants (6F), aged 19-38, from a large university campus. Participants included undergraduate students, graduate students, and administrative staff from a range of departments (e.g., Computer Science, Engineering, Business, and Biology). We pre-screened participants to ensure that they did not have prior 3D modeling experience. The expert role was taken by a member of the research team. We decided to use the same expert for all participants and conditions to maintain consistency across the responses and the type of help provided.

A. Study Design

We used a within-subject design to minimize the impact of the known high variation among participants. To eliminate order effects, we randomized the order of the In-Context Chat and External Chat conditions using a Latin Square. Following the study task, participants were interviewed by the researchers.

Similar to our observational study, participants were asked to create 3D models to match a reference image. Their tasks were to construct two 3D models of castles (Fig. 2), which we determined to have approximately equal difficulty through pilot tests. The two castles were presented in random order.

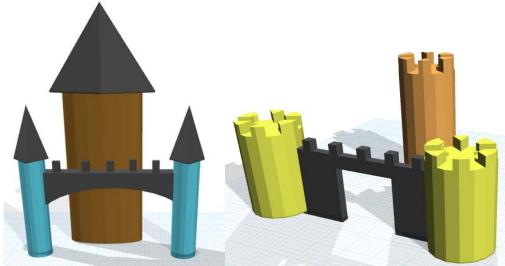


Fig. 2. Castles used in the two task conditions

For the External Chat condition, participants communicated with the expert via an external application. We chose *HipChat*, a web-based chat tool, as it had the required features but is not

widely used. As in the observational study, novice participants were encouraged to ask questions of the expert through HipChat and were able to type questions and share screenshots (Fig. 4).

Our study considered the following measures for each participant: 1) Task performance, including the amount of the reference model the participant was able to complete; 2) help-seeking strategies, including how participants asked questions and had conversations with the expert; and 3) participants' subjective assessment of the two MarmalAid variants.

We conducted the study in an enclosed lab using a desktop computer running Windows 7 and the Chrome web browser.

B. Procedure

Each study session comprised four parts. For the first 10 minutes, participants completed the consent form and a demographic questionnaire, and followed a printed tutorial document that described MarmalAid's key features.

Next, participants completed the 3D modeling task for the two conditions (presented in counterbalanced order). Prior to the In-Context Chat condition, we showed participants how to use MarmalAid's Q&A feature and how to share their view with the expert. Prior to the External Chat condition, we showed participants how to use HipChat and how to share screenshots with the expert. The expert, portrayed by a member of our research team, was available throughout both of the task conditions. Participants were given 20 minutes to complete each modeling task.

Finally, participants answered questions about their experience in a 10-minute follow-up interview. The total study session took about 60-70 minutes. We compensated each participant with a \$20 Amazon gift card.

C. Data Collection and Analysis

Throughout the study session, we recorded the participant's screen and audio recorded their interview responses. We collected all of the models created during each study session. During the two study tasks, we recorded usage of MarmalAid and chat transcripts through time-stamped activity logs.

To measure participants' performance on the study tasks, we analyzed the 3D models created during each of the study tasks. Each model was scored by a 3D modeling expert along two axes: accuracy of the model to the original reference image, and completion of the model. The expert scored each model on a separate 10-point scale for accuracy and for completion. Accuracy was scored based on use of the same primitive shapes, similar positioning of the shapes, and the characteristics of the shapes used (color, size, and aspect ratio). Completion was scored based on the number of components completed (castle base, pillars, walls).

To understand participants' help-seeking behavior across the study tasks, we collected and analyzed the text of the questions asked by the participants. We performed qualitative coding on each question to determine whether it was related to the 3D modeling task itself (e.g., "how can I combine objects to create a cone?") or to the use of the UI (e.g., "is there a copy and paste feature?"). Finally, to understand participants' subjective responses to using the two prototypes, we analyzed the post-task interviews. We used a bottom-up inductive analysis approach to identify broader themes within the participants' feedback.

D. Results

Task Completion and Accuracy: We designed our modeling tasks so that participants would be able to continue the modeling task for at least 20-minutes, regardless of their talent for creating 3D models. Indeed, no participant completely finished either of the modeling tasks during the 20-minute task period. On average, participants completed about 30% of each of the castle models, resulting in an average completeness score of 3. There was no significant difference of completion rate across two study conditions, according to a paired-sample t-test ($t=-0.55$, $p=0.59$, two-tailed) and we did not observe any order effects.

Regarding the accuracy of the completed models, the models created in the In-Context Chat condition were marginally more accurate than models created in the External Chat condition, with an average accuracy score of 5.8 vs. 5.5, respectively. However, this difference in accuracy was not statistically significant ($t=1.07$, $p=0.31$, two-tailed) and there were no order effects.

Questions Asked: Overall, participants asked more questions during the In-Context Chat condition than in the External Chat condition, averaging 6.01 vs. 5.10 questions, respectively. This difference was significant ($t=2.24$, $p\leq 0.05$, two-tailed).

We analyzed participants' to determine whether there were more questions about how to create the 3D model or about how to operate the modelling UI. We found that the majority of questions in all conditions were about how to perform actions within the 3D modeling task, rather than about operating the UI. In fact, on average, participants asked more than 4 times as many questions about 3D modeling than about the UI, averaging 4.55 and 0.97 questions per study session, respectively. This difference was significant ($t=3.92$, $p\leq 0.01$, two-tailed).

Additionally, study condition seemed to affect the types of questions asked. Participants asked more 3D modeling questions during the In-Context Chat condition than in the External Chat condition, averaging 4.90 vs. 3.90 questions, respectively. This difference was significant ($t=2.34$, $p\leq 0.05$, two-tailed). This finding suggests that participants may have been more focused on the 3D modeling task when using the in-context help chat.

We analyzed the topics of questions in each category. The 3D modeling questions included questions about manipulating the 3D workspace (i.e., the workplane), how to compose complex shapes, and how to create holes. With the brown and blue castle (Fig. 3, left), the top question was usually about creating a cone, as MarmalAid did not offer a cone primitive. For the yellow and brown castle (Fig. 3, right), the majority of participants asked about how to create a window through the castle wall, as they were not experienced in using holes.

UI questions included asking for explanations of how certain features worked, such as object grouping, and questions about whether certain features were available, such as copy-and-paste.

Conversational Structure: We analyzed the conversations across each study condition both from the perspectives of novices asking questions and the expert providing answers.

A key observation that we made was that the number of clarification questions asked by the expert varied across the study conditions. Although participants in the In-Context Chat condition asked more questions about 3D modeling, the expert asked

fewer follow-up questions in the In-Context Chat condition than in the External Chat condition, averaging 5.18 vs 3.09 questions, respectively. This difference was significant ($t=-2.24$, $p\leq 0.05$, two-tailed), suggesting that participants may have asked fewer ambiguous questions when using In-Context Chat.

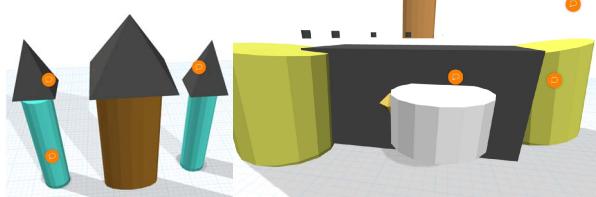


Fig. 3. Castle models created during the Internal Chat condition (left from P01; right from P03). The orange spheres indicate several chat windows at different locations.

In the External Chat condition, participants sometimes used screenshots to provide additional context for their questions (Fig. 4). On average, participants shared 2 screenshots per study task in this condition.

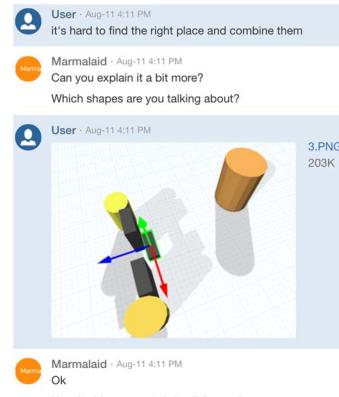


Fig. 4. Example conversation from the External Chat condition. The novice's initial question is considered ambiguous by the expert, and the expert requests a screenshot to provide further context for the question.

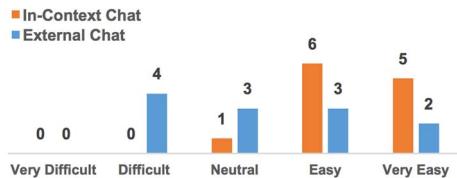
The length of conversations also differed between conditions. Participants wrote more words in the In-Context Chat condition than in the External Chat condition, averaging 336.45 words vs. 269.35 words, respectively. This difference was significant ($t=-2.55$, $p\leq 0.05$, two-tailed). This difference in the length of conversations may reflect greater engagement from participants when using in-context help. Longer conversations in the In-Context Chat condition may also be influenced by MarmalAid's ability to create multiple simultaneous chat windows. Nearly all participants (10 out of 12) in the In-Context Chat condition created multiple chat windows, creating three windows on average (examples shown in Fig. 3).

Subjective Feedback: At the end of each study condition, we asked participants to rate their experiences and preferences. Participants' ratings on Likert-style questions indicate that participants significantly preferred In-Context Chat ($w=37$, $p\leq 0.05$), and found In-Context Chat to be more useful ($w=27$, $p\leq 0.01$). Participant responses are shown in Fig. 5.

We asked participants to rate the difficulty of modeling in each condition. On a scale of *Very Difficult* to *Very Easy*, most participants commonly rated the modeling tasks as *Somewhat Difficult*, which is not surprising given that this was the first time the participants had created 3D models. A Wilcoxon signed-rank

test found no difference between the perceptions of difficulty under both conditions ($w=6$, $p=0.33$).

How would you describe the **overall process** of getting help?



How would you describe the **overall usefulness** of getting help?

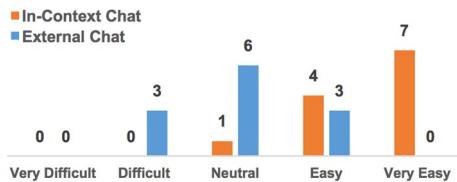


Fig. 5. Participants' ratings for overall preference and usefulness.

User Perspectives of Help Quality: We asked participants about task difficulty, and what they liked and disliked about each user interface. Participants pointed out that they had difficulty in assembling objects, and in navigating around the different views of the 3D model. When seeking help, participants noted that they lacked the correct vocabulary to ask their questions.

Participants noted several advantages to MarmalAid's real-time, in-context help. First, they noted that in-context help created a shared visual context that made it easier to ask questions:

P04: I found the interaction here [in-context chat] to be more intuitive and straight-forward because with the other [external] chat, I had to explain exactly what I meant and ... it was hard to get the exact shape across ... for example, with the castle ridges on top, I didn't know how to explain that, so I had to be creative and explain it like a chess piece...made it more difficult.

The ability to anchor questions on a 3D model enabled participants to direct the expert's attention, and allowed the novice and expert to separate out different discussion threads:

P11: When you set up a chat for one particular shape, it implicitly implies that the context is for this shape...so I like that you can sort of create a specific zone [for the questions]

Sometimes, the lack of shared visual context resulted in miscommunication in the chat between the novice and the expert:

P12: I was trying to [explain] how to create the smokes on the cylinder...ended up with the option of making holes with the square...that wasn't the desired result because it was difficult to explain what I was trying to do [in HipChat] ...I ended up with something exactly opposite of what I wanted...just frustrating.

Participants noted several additional benefits of the in-context chat, such as reducing the need for context switching and receiving notifications as soon as the expert replied:

P08:[With HipChat] I had to stop my train of thought, go to another tab, maybe take a screenshot, and then attach it...sometimes the expert would be waiting for me to say something and I would just be on the other tab...

MarmalAid's in-context chat enabled participants to ask questions without switching tasks. Also, because MarmalAid notified users as soon as the expert provided a response, participants did not miss out on responses from the expert. In contrast, we observed that five participants neglected to look at the chat

window during the External Chat condition, even after the expert had provided a response.

Suggested Improvements: Participants suggested several additional features for future versions of MarmalAid, including the ability to ask voice-based questions, features for direct annotations on the model, and the ability to selectively share parts of a model, rather than sharing the entire model.

While ten of the twelve participants preferred MarmalAid's in-context chat to the external chat, two preferred the external chat. Both of these participants used external chat for their second modeling task and stated that they asked for more assistance with external chat. One of these participants mentioned that in-context chat window felt out of place in the modeling tool and took up too much screen space. The other participant noted that chat windows took up screen space even after the questions were answered. While the In-Context Chat condition allowed users to minimize or close chat windows, we encouraged participants to retain their chat windows throughout the study task.

Reuse Value and Broader Applicability: During the interviews, most participants (10 out of 12) said that they would be likely to keep using MarmalAid's in-context Q&A feature if it were available. Participants also suggested that MarmalAid's feedback model could be extended to other types of applications such as image editing programs, statistics tools, and programming environments. Participants also suggested that this type of collaborative help could be useful when working collaboratively, such as when soliciting feedback on a design project.

One participant compared the ability to work with an expert in MarmalAid to following along with an expert in online videos, but noted that the novice cannot directly ask for help when watching a video, as they can with MarmalAid:

P05: YouTube is good so you can mimic what they [experts] are doing... but, you can have differences in version and speed and understanding... [experts] can use shortcuts or use functions which you may not know...with real-time [help], the expert can really break it down for you.

Participants noted that MarmalAid's in-context chat can benefit novices by making it easier to ask questions and access experts, but also that MarmalAid would make it easier to provide help when taking on the expert role.

VII. DISCUSSION

This paper contributes a new design for real-time, in-context help for 3D modeling, and provides empirical insights into how novice 3D modelers use and perceive remote real-time expert help via real-time, in-context chat vs. an external chat application and how it affects their 3D learning tasks. We now reflect on key lessons learned from this research and design opportunities in HCI for improving real-time, in-context help systems.

A. Supporting 3D Modeling with Real-Time, In-Context Q&A

Participants in our study encountered various challenges in learning 3D modeling, similar to those discussed in prior work (e.g., [4,17,21,27,37]), including problems in understanding 3D geometry and developing an appropriate domain-related vocabulary. Although our findings were not conclusive about whether real-time, in-context help provides any immediate performance improvement for novice 3D modelers, our findings suggest that in-context help enables novices to spend less time clarifying

questions and more time asking additional questions. There is clear indication that users can ask more learning-related questions when they have to spend less time on explaining their context to the helper. Furthermore, users strongly preferred the in-context chat experience and found it more useful than external chat, in part because they were able to localize conversations with experts at the relevant location within a 3D model.

Our findings have immediate applicability to improving help in settings such as 3D modeling workshops, classrooms, makerspaces, and online design sharing and troubleshooting forums (e.g., *Thingiverse* [1]), where novice 3D modelers already work with more experienced users to solve problems and learn about new features. Future work can tease out the learning and performance benefits of in-context help through long-term field deployments of MarmalAid in such settings and can explore the differences between novice and more expert users. In the future, we may be able to automatically assign certain helpers based on their skills and their experience working on similar problems. Since MarmalAid's features mostly support primitive 3D modeling tasks, it would be useful to expand the scope of the application to support more polygon-oriented 3D shape modeling, and to assess the usefulness of in-context, real-time help for more advanced 3D modeling tasks.

B. Design Opportunities for Real-Time, In-Context Help

Although our main goal in this work was to understand and design remote real-time help to support 3D modeling tasks, our participants expressed enthusiasm for expanding this approach to other application domains. Below we discuss opportunities in to further develop and expand the idea of real-time, in-context help for tasks beyond 3D modeling.

Allowing for domain-related queries in-context: A key finding from our study was that participants asked a larger proportion of questions about 3D modeling when using real-time, in-context help (rather than asking questions about the UI). This differs from many previous in-context help systems (e.g., [9,16,20,29]) that focused on teaching the user interface rather than the domain of the task. Our findings suggest that more work is needed to understand users' preferences in soliciting domain-specific contextual help. In particular, participants in our study liked to point at objects when asking a question to direct attention to that area. This notion of situated questions could apply in other domains, but would likely differ significantly in tasks such as programming where contextual help needs are very different [6]. In the future, we may develop approaches to automatically detect "location" across different task domains.

Expanding the scope of on-screen referents: Another lesson we learned from this work was that users appreciate being able to anchor their questions to a specific problem area, but that some questions may reference multiple objects. How can users describe their referents in these situations? Participants in our study suggested adding annotation tools that would enable them to point at various items on screen, and to explain their problems at different levels of granularity. Future work can explore what on-screen referents are needed for more complex 3D scenes, as well as how these referents can be automatically detected. For example, LemonAid [9] can detect on-screen referents automatically and attaches crowdsourced Q&A discussion if a white list of existing UI labels is initially provided—what would be the

input for automatic detection for 3D geometry or other types of applications, such as programming tools?

Scaling up real-time help requests: Our current work only tests real-time help between one novice and one expert. How can this approach scale to larger groups? One approach may be to combine real-time Q&A user interfaces with emerging work in real-time crowdsourcing [25,26]. Another approach may be to cache requests or save previously asked Q&A. It may be worth exploring how real-time help could be combined with asynchronous help, such as by viewing questions from other users, as has been explored in crowdsourced contextual help tools (e.g., [9,29]).

Automating real-time help: Another key research opportunity is to design better automated real-time help systems, such as chatbots. Our findings imply that in-context chatbots would be perceived as useful in learning contexts as users would be able to better describe their help needs (requiring less back-and-forth). Future work can explore how learners can be supported by automated approaches that combine synchronous help technologies [28], intelligent tutoring systems [32], and context-aware help. Furthermore, given the emergence of customer-service chatbots [22,43], future work can explore the challenges and opportunities of localizing these chatbots in context of a given application or informational website.

Supporting design feedback and collaboration: Several participants noted that MarmalAid's real-time, in-context conversation features could also be useful for soliciting feedback or working collaboratively on design problems. Although collaborative CAD systems for professionals has been explored previously [13], there is opportunity to further explore how in-context communication that allows users to localize their conversations within the interface can impact collaboration. In particular, there may be a benefit of this type of communication for virtual teams [2], where establishing a shared context can pose communication challenges. In addition, not all collaborative communication needs to occur in real-time; the in-context Q&A can also be used asynchronously and augment the history of design decisions.

Limitations: While all of our participants were novice modelers, our sample size was small, and we did not control for other individual differences, such as experience with different types of software or learning styles. Although we worked with 3D modeling experts to develop test tasks suitable for novices, it is possible that participants' behaviors could be different if they had more time or if they had worked on different 3D modeling tasks. Finally, our prototype was tested on an experimental application with only basic features; some caution should be used when generalizing results to other 3D modeling applications.

CONCLUSIONS

In this paper, we have investigated how novice 3D modelers can use remote, real-time expert help during their initial learning tasks. We have introduced MarmalAid, a web-based 3D modeling tool that allows users to visually share their context and obtain in-application real-time help from another user. The key innovation of MarmalAid is that users can have real-time conversations with experts that is anchored to the 3D geometry of their models. Our empirical findings lead to several design opportunities in HCI for using real-time, in-context help in applications beyond 3D modeling, and inform the design of automated real-time help systems.

REFERENCES

- [1] Celena Alcock, Nathaniel Hudson, and Parmit K. Chilana. 2016. Barriers to Using, Customizing, and Printing 3D Designs on Thingiverse. In *Proceedings of the 19th International Conference on Supporting Group Work*, 195–199.
- [2] Gary Baker. 2002. The effects of synchronous collaborative technologies on decision making: A study of virtual teams. *Information Resources Management Journal* 15, 4: 79.
- [3] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. 2010. Information needs in bug reports: improving cooperation between developers and users. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, 301–310.
- [4] Erin Buehler, Shaun K. Kane, and Amy Hurst. 2014. ABC and 3D: opportunities and obstacles to 3D printing in special education environments. In *Proceedings of the 16th international ACM SIGACCESS conference on Computers & accessibility*, 107–114.
- [5] Andrea Bunt, Patrick Dubois, Ben Lafreniere, Michael A. Terry, and David T. Cormack. 2014. TaggedComments: promoting and integrating user comments in online application tutorials. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, 4037–4046.
- [6] Yan Chen, Sang Won Lee, Yin Xie, YiWei Yang, Walter S. Lasecki, and Steve Oney. 2017. Codeon: On-Demand Software Development Assistance. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, 6220–6231.
- [7] Pei-Yu Chi, Sally Ahn, Amanda Ren, Mira Dontcheva, Wilmot Li, and Björn Hartmann. 2012. Mixt: automatic generation of step-by-step mixed media tutorials. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*, 93–102.
- [8] Parmit K. Chilana, Tovi Grossman, and George Fitzmaurice. 2011. Modern software product support processes and the usage of multimedia formats. In *Proceedings of the 2011 annual conference on Human factors in computing systems*, 3093–3102.
- [9] Parmit K. Chilana, Andrew J. Ko, and Jacob O. Wobbrock. 2012. LemonAid: Selection-based Crowdsourced Contextual Help for Web Applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12)*, 1549–1558.
- [10] A. Crabtree, J. O'Neill, P. Tolmie, S. Castellani, T. Colombino, and A. Grasso. 2006. The practical indispensability of articulation work to immediate and remote help-giving. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, 228.
- [11] D. K Farkas. 1993. The role of balloon help. *ACM SIGDOC Asterisk Journal of Computer Documentation* 17, 2: 3–19.
- [12] Jennifer Fernquist, Tovi Grossman, and George Fitzmaurice. 2011. Sketch-sketch revolution: an engaging tutorial system for guided sketching and application learning. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, 373–382.
- [13] Jerry YH Fuh and W. D. Li. 2005. Advances in collaborative CAD: the state-of-the art. *Computer-Aided Design* 37, 5: 571–581.
- [14] Susan R. Fussell, Robert E. Kraut, and Jane Siegel. 2000. Coordination of communication: effects of shared visual context on collaborative work. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, 21–30.
- [15] Floraine Grabler, Maneesh Agrawala, Wilmot Li, Mira Dontcheva, and Takeo Igarashi. 2009. Generating photo manipulation tutorials by demonstration. *ACM Transactions on Graphics (TOG)* 28, 3: 66.
- [16] Tovi Grossman and G. Fitzmaurice. 2010. Toolclips: An investigation of contextual video assistance for functionality understanding. In *Proceedings of the 28th international conference on Human factors in computing systems*, 1515–1524.
- [17] Tovi Grossman, George Fitzmaurice, and Ramtin Attar. 2009. A survey of software learnability: metrics, methodologies and guidelines. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 649–658.
- [18] Grossman, Tovi, Matejka, Justin, and Fitzmaurice, George. 2010. Chronicle: Capture, Exploration, and Playback of Document Workflow Histories. In *ACM Symposium on User Interface Software & Technology*.
- [19] Susan M. Harrison. 1995. A comparison of still, animated, or nonillustrated on-line help with written or spoken instructions in a graphical user interface. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, 82–89.
- [20] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. 2010. What Would Other Programmers Do: Suggesting Solutions to Error Messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*, 1019–1028.
- [21] Nathaniel Hudson, Celena Alcock, and Parmit K. Chilana. 2016. Understanding Newcomers to 3D Printing: Motivations, Workflows, and Barriers of Casual Makers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*.
- [22] Shep Hyken. 2017. AI And Chatbots Are Transforming The Customer Experience. *Forbes*. Retrieved from <https://www.forbes.com/sites/shephyken/2017/07/15/ai-and-chatbots-are-transforming-the-customer-experience>
- [23] Caitlin Kelleher and R. Pausch. 2005. Stencils-based tutorials: design and evaluation. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, 541–550.
- [24] Benjamin Lafreniere, Tovi Grossman, and George Fitzmaurice. 2013. Community Enhanced Tutorials: Improving Tutorials with Multiple Demonstrations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*, 1779–1788.
- [25] Walter S. Lasecki, Kyle I. Murray, Samuel White, Robert C. Miller, and Jeffrey P. Bigham. 2011. Real-time crowd control of existing interfaces. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, 23–32.
- [26] Walter S. Lasecki, Rachel Wesley, Jeffrey Nichols, Anand Kulkarni, James F. Allen, and Jeffrey P. Bigham. 2013. Chorus: a crowd-powered conversational assistant. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, 151–162.
- [27] Ghang Lee, Charles M. Eastman, Tarang Taunk, and Chun-Heng Ho. 2010. Usability principles and best practices for the user interface design of complex 3D architectural design and engineering tools. *International Journal of Human-Computer Studies* 68, 1–2: 90–104.
- [28] Olivera Marjanovic. 1999. Learning and teaching in a synchronous collaborative environment. *Journal of Computer Assisted Learning* 15, 2: 129–138.
- [29] Justin Matejka, Tovi Grossman, and George Fitzmaurice. 2011. IP-QAT: In-product Questions, Answers, & Tips. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*, 175–184.
- [30] Andrew Milne, Bernhard Riecke, and Alissa Antle. Exploring Maker Practice: Common Attitudes, Habits and Skills from Vancouver's Maker Community. *Studies* 19, 21: 23.
- [31] S. Palmiter, J. Elkerton, and P. Baggett. 1991. Animated demonstrations vs written instructions for learning procedural tasks: a preliminary investigation. *International Journal of Man-Machine Studies* 34, 5: 687–701.
- [32] Martha C. Polson and J. Jeffrey Richardson. 2013. *Foundations of intelligent tutoring systems*. Psychology Press.
- [33] Suporn Pongnumkul, Mira Dontcheva, Wilmot Li, Jue Wang, Lubomir Bourdev, Shai Avidan, and Michael F. Cohen. 2011. Pause-and-play: automatically linking screencast video tutorials with applications. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, 135–144.
- [34] Erika Shehan Poole, Marshini Chetty, Tom Morgan, Rebecca E. Grinter, and W. Keith Edwards. 2009. Computer help at home: methods and motivations for informal technical support. In *Proceedings of the 27th international conference on Human factors in computing systems*, 739–748.
- [35] Erika Shehan Poole, W. Keith Edwards, and Lawrence Jarvis. 2009. The Home Network as a Socio-Technical System: Understanding the Challenges of Remote Home Network Problem Diagnosis. *Comput. Supported Coop. Work* 18, 2–3: 277–299.
- [36] Kimberly Sheridan, Erica Rosenfeld Halverson, Breanne K Litts, Lisa Brahms, Lynette Jacobs-Priebe, and Trevor Owens. Learning in the Making: A Comparative Case Study of Three Makerspaces - ProQuest. *Harvard Educational Review* 84, 4, R.

- [37] Rita Shewbridge, Amy Hurst, and Shaun K. Kane. 2014. Everyday Making: Identifying Future Uses for 3D Printing in the Home. In *Proceedings of the 2014 Conference on Designing Interactive Systems* (DIS '14), 815–824.
- [38] Piyawadee Sukaviriya, Ellen Isaacs, and Krishna Bharat. 1992. Multimedia help: a prototype and an experiment. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, 433–434.
- [39] Michael Twidale and Karen Ruhleder. 2004. Where am I and who am I?: issues in collaborative technical help. In *ACM conference on Computer supported cooperative work*, 378–387.
- [40] Laton Vermette, Shruti Dembla, April Y. Wang, Joanna McGrenere, and Parmit K. Chilana. 2017. Social CheatSheet: An Interactive Community-Curated Information Overlay for Web Applications. *Proc. ACM Hum.-Comput. Interact.* 1, CSCW: 1–19.
- [41] Ron Wakkary, Markus Lorenz Schilling, Matthew A. Dalton, Sabrina Hauser, Audrey Desjardins, Xiao Zhang, and Henry WJ Lin. 2015. Tutorial authorship and hybrid designers: The joy (and frustration) of DIY tutorials. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, 609–618.
- [42] Tricia Wang and Joseph “Jofish” Kaye. 2011. Inventive Leisure Practices: Understanding Hacking Communities As Sites of Sharing and Innovation. In *CHI ’11 Extended Abstracts on Human Factors in Computing Systems* (CHI EA ’11), 263–272.
- [43] Anbang Xu, Zhe Liu, Yufan Guo, Vibha Sinha, and Rama Akkiraju. 2017. A New Chatbot for Customer Service on Social Media. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, 3506–3510.

ZenStates: Easy-to-Understand Yet Expressive Specifications for Creative Interactive Environments

Jeronimo Barbosa

*IDMIL, CIRMMT,**McGill University**Montreal, Canada*

jeronimo.costa@mail.mcgill.ca

Marcelo M. Wanderley

*IDMIL, CIRMMT**McGill University & Inria**Montreal, Canada*

marcelo.wanderley@mcgill.ca

Stéphane Huot

*Inria**Univ. Lille, UMR 9189 - CRISTAL**Lille, France*

stephane.huot@inria.fr

Abstract—Much progress has been made on interactive behavior development tools for expert programmers. However, little effort has been made in investigating how these tools support creative communities who typically struggle with technical development. This is the case, for instance, of media artists and composers working with interactive environments. To address this problem, we introduce ZenStates: a new specification model for creative interactive environments that combines Hierarchical Finite-States Machines, expressions, off-the-shelf components called *Tasks*, and a global communication system called the *Blackboard*. Our evaluation is three-folded: (a) implementing our model in a direct manipulation-based software interface; (b) probing ZenStates’ expressive power through 90 exploratory scenarios; and (c) performing a user study to investigate the understandability of ZenStates’ model. Results support ZenStates viability, its expressivity, and suggest that ZenStates is easier to understand-in terms of decision time and decision accuracy-compared to two popular alternatives.

Index Terms—Human-computer interaction; hierarchical finite state machines; creativity-support tools; interactive environments; end-user programmers.

I. INTRODUCTION

Over the last years, several advanced programming tools have been proposed to support the development of rich interactive behaviors: The HsmTk [1] and SwingStates [2] toolkits replace the low-level event-based paradigm with finite-states machines; ConstraintJS [3] introduces constraints declaration and their implicit maintenance as a way to describe interactive behaviors; InterState [4] and Gneiss [5] are dedicated programming languages and environments. These advances are primarily focused on programmers and are important because:

- They can make interactive behavior *easier to understand*—sometimes even by non-experts programmers. This is the case, for instance, of the SwingStates toolkit [2]. SwingStates was successfully used by graduate-level HCI students to implement advanced interaction techniques despite of their limited training, when other students with similar skills were unsuccessful in implementing the same techniques with standard toolkits. In addition,

This work was partially supported by the NSERC Discovery Grant and the *Fonds de recherche du Québec–Société et culture* (FRQSC) research grant.

978-1-5386-4235-1/18/\$31.00 ©2018 IEEE

better understanding of the implementation of interactive behaviors can make it easier to reuse and modify [1];

- These tools do not sacrifice *expressiveness* to make programming faster or understanding easier. Examples such as [3] and [5] illustrate how such tools can potentially implement a wide variety of complex interactive behaviors, which would have been more costly in existing alternatives.

Despite these advantages, little effort has been made to investigate how these advances could support end-user programmers (EUP) [6], [7] from creative communities who typically struggle with coding such as artists and designers [8], [9]. Making these behaviors *easier to understand* could make them easier to learn, reuse, and extend. At the same time, addressing *expressiveness* (i.e., going beyond standard and well-defined behaviors) could improve the diversity of creative outcomes, helping in the exploration of alternatives. All in all, these characteristics have the potential to foster one’s creative process [10].

One example of such community is composers and media artists working with *creative interactive environments* [11]. These environments are immersive pieces that react to the presence of visitors, based on a wide diversity of input sensors (eg. video cameras) and actuators (eg. sound, lighting systems, video projection, haptic devices). Creating easy to understand and expressive development tools for these environments is relevant and challenging for two reasons. First, interactive environments’ setups are often more complex than standard interactive systems (such as desktop or mobile computers), as they (i) need to deal with multiple advanced input/output devices and multiple users that both can connect/disconnect/appear/disappear dynamically; (ii) need to be dynamic, flexible and robust. Therefore, by tackling more complex setups, some of its unique features could potentially transfer to more standard setups (e.g. desktop). Secondly, because programming directly impacts on the artistic outcomes, increasing programming abilities and possibilities can likely yield finer artistic results, by reducing the gap between artistic and technical knowledge.

These are the context and the challenge we address in this paper. Here, we investigate innovations brought by powerful development tools for expert programmers [4], [5], [12],



Fig. 1. ZenStates allows users to quickly prototype interactive environments by directly manipulating *States* connected together via logical *Transitions*. *Tasks* add concrete actions to states and can be fine-tuned via high-level parameters (e.g. ‘intensity’ and ‘pan’). *Blackboard* global variables (prefixed with the ‘\$’ sign) can enrich tasks and transitions.

aiming at bringing them to the context of creative interactive environments. The result is ZenStates: an easy to understand yet expressive specification model that allows creative EUPs to explore these environments using Hierarchical Finite-State Machines, expressions, off-the-shelf components called tasks, and a global communication system called the blackboard. This model has been implemented in a visual direct manipulation-based interface that validates our model (Fig. 1). Finally, as an evaluation, we also report: (a) 90 exploratory scenarios to probe ZenStates’ expressive power; and (b) a user study comparing ZenStates’ ease of understanding against the specification model used by two popular interactive environments development tools: Puredata and Processing.

II. RELATED WORK

One of the most basic approaches for interactive environments development is trigger-action programming [13]. This approach—used, for instance, in the context of smart homes—describes simple conditional “if-then-else” behaviors that can be easily customized by users. Despite being intuitive and effective, this approach limits interactive environments to simple one-level behaviors. For instance, any example involving temporal actions cannot be implemented.

Other approaches can be found within the context of creative communities. This is the case of the visual “Max paradigm” [14], found in musical tools such as Max/MSP, and Puredata. These dataflow-inspired software environments have become widely adopted over the years by creative EUPs. They allow users to build multimedia prototypes by visually combining elementary building blocks, each performing a specific real-time operation. However, because everything is designed for real-time, little support is provided for managing the flow of control of application (e.g. conditionals, loops, routines) [14].

Another approach is the case of textual programming environments and libraries [15], such as Processing, openFrame-

works, and Cinder. However, although these tools lower the floor required to get started in programming, they still require users to deal with abstract concepts whose meaning is not often transparent, especially to novices [16].

To overcome this understandability issue, direct manipulation [17] has been applied in this context to make programming less abstract and therefore more tangible to creative users. Examples include Sketch-N-Sketch [18]—where direct manipulation is combined with textual programming language—and Para [19]—direct manipulation combined with constraint-based models. While the strategy is helpful, the solution is kept on the interface level (i.e., that could potentially be applied to every specification model), whereas the specification model itself remains unexplored. Furthermore, both examples are limited to static artistic results (i.e., not capable of reacting to inputs over time).

A. Non-programmers expressing interactive behaviors

Given our interest in exploring more accessible specification models, we need to understand first how non-programmers deal with expressing interactive behaviors. A few works have dealt with this topic.

In [20] for instance, two studies focused on understanding how non-programmers express themselves in solving programming-related problems. In one study, 10-11 years old children were asked to describe how they would instruct a computer to behave in interactive scenarios of the Pac-man game. In another, university-level students were asked to do the same in the context of financial & business analytical problems. Results suggest that participants tended to use an event-based style (e.g., if then else) to solve their problems, with little attention to the global flow of the control (typical in structured imperative programming languages, for example). This approach to problem-solving arguably has similarities to the approach that state machines have in problem-solving.

Some work also presented practices and challenges faced by professional multimedia designers in designing and exploring rich interactive behaviors derived from series of interviews and a survey [8]. The results suggest that in the case of designing interactive behaviors: a) current tools do not seem to fulfill the needs of designers, especially in the early stages of the design process (i.e., prototyping); b) texts, sketches, and several visual “arrow-centered” techniques (e.g. mind map, flowcharts, storyboards) are among the most used tools to communicate ideas; and c) designers prefer to focus on content rather than “spending time” on programming or learning new tools. These findings resulted in a new system, DEMAIS, which empowers designers to communicate and do lo-fi sketch-based prototypes of animations via interactive storyboards.

Similar findings were reported by [9]. In a survey with 259 UI designers, the authors found that a significant part of participants considered that prototyping interactive behaviors was harder than prototyping appearance (i.e., the “look and feel”). Participants reported struggling with communicating interactive behaviors to developers—although this communication was necessary in most of the projects—when compared to

communicating appearance. In addition, participants reported necessity for iteration and exploration in defining these behaviors, which generally involved dealing with changing states. The authors conclude by reporting the need for new tools that would allow quicker and easier communication, design and prototyping of interactive behaviors.

These works suggest that state machines could be suited to the development of interactive behavior by non-programmers, and, we believe, by creative end-user programmers who struggle with technical development.

B. Experts programming interactive behaviors

Several tools have been developed to support expert programming of interactive behaviors [21]. Examples include Gneiss [5], ICon [22], and OpenInterface [23]. Among these, a significant number of tools use state machines to specify such interactive behaviors. A complete survey is beyond the scope of this research. Here, we focus on works we consider more closely related to our research.

Perhaps one of the most well-known example is [12], which introduced StateCharts: a simple yet powerful visual-based formalism for specifying reactive systems using enriched hierarchical state machines. StateCharts' formalism was later implemented and supported via a software tool called StateMate [24], and is still used (more than 30 years later) as part of IBM Rational Rhapsody systems for software engineers¹.

Examples are also notable in the context of GUIs, for instance: HsmTk [1], a C++ toolkit that incorporates state machines to the context of rich interaction with Scalable Vector Graphics (SVG); SwingStates [2], that extends the Java's Swing toolkit with state machines; and FlowStates [25], a prototyping toolkit for advanced interaction that combines state machines for describing the discrete aspects of interaction, and data-flow for its continuous parts.

More recently, research has been done on how to adapt state machines to the context of web applications. ConstraintJS [3] proposed an enhanced model of state machines that could be used to control temporal constraints, affording more straightforward web development. This idea is further developed by the authors in [4], resulting in InterState, a new full programming language and environment that supports live coding, editing and debugging.

These works introduce advances that make interactive behaviors programming faster and easier to understand. However, as a drawback, these tools are hardly accessible for creative EUPs who would still need to develop strong programming skills before benefiting from them. To address this problem, we have built upon these works, aiming at making them accessible to creative EUPs. The result, a new specification model called *ZenStates*, is introduced in the next section.

III. INTRODUCING ZENSTATES

ZenStates is a specification model centered on the idea of *State machines*. A *State Machine* is defined as a set of abstract

States to which users might add *Tasks* that describe (in terms of *high-level parameters*) what is going to happen when that particular state is being executed. These states are connected to one another via a set of logical *Transitions*. Execution always starts at the “Begin” state, following these transitions as they happen until the end. At any moment, inside a Task or a Transition, users can use *Blackboard* variables to enrich behaviors (e.g., use the mouse x position to control the volume, or to trigger certain transitions).

A. Enriched state machines as specification model

ZenStates borrows features from the enriched state machines model proposed by StateChart [12] and StateMate [24]. These works overcome typical problems of state machines—namely, the exponential growth of number of states, and its chaotic organization—by introducing:

- **Nested state machines (clustering):** One state could potentially contain other state machines;
- **Orthogonality (parallelism):** Nested state machines need to be able to execute independently and in parallel whenever their parent state is executed;
- **Zooming-in and zooming-out:** A mechanism that allows users to navigate between the different levels of abstraction introduced by the nesting of state machines;
- **Broadcast communication:** That allows simple event messages to be broadcasted to all states, having the potential to trigger other states inside the machine.

When combined, these features result in a powerful hierarchical model of state machines. This model provides improved *organization* (i.e., state machines are potentially easier to understand), *modularity* (i.e., subparts of a system could be independently designed and later merged into a larger structure), and *expressiveness* (broadcast communication allows enhanced description of complex behaviors) when compared to the traditional states machine approach.

B. Key features

Building upon this model, *ZenStates* proposes five key features described in the following subsections.

1) Extending communication: the Blackboard

The blackboard—in the top-right of Figure 2—is a global repository of variables that can be defined and reused anywhere inside a state machine (i.e., inside states, nested states, tasks or transitions). Therefore, we extend the notion of broadcast communication introduced by [12] because variables can be used to other functionalities, and not only triggering transitions. In addition, because the blackboard is always visible on the interface, users can easily infer what inputs are available on the environment, as well as their updated values.

Users can interact with the blackboard using *pull* and *push* operations. Pull operations are accessed by using the variable's name (as in Gneiss [5]). Push operations can be performed by tasks (see next subsection).

Some useful context-dependent variables can be automatically added to the blackboard and become directly accessible to users. In interactive environments, examples include mouse

¹<https://www.ibm.com/us-en/marketplace/rational-rhapsody>

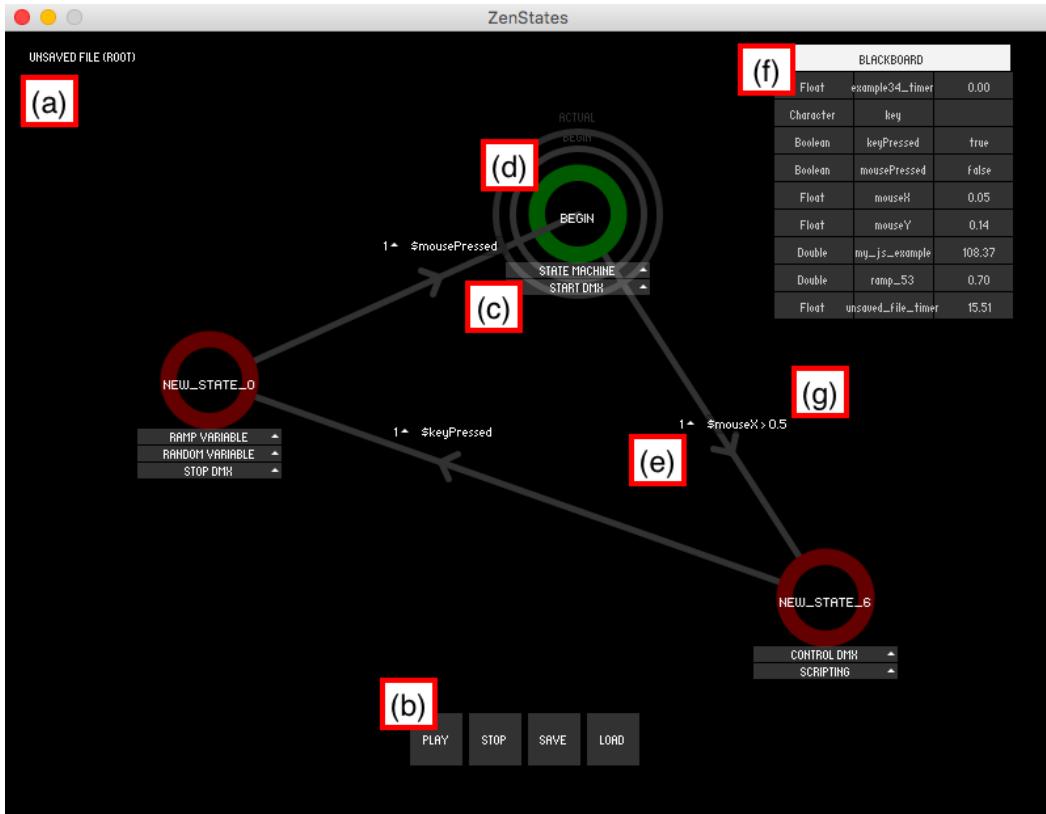


Fig. 2. ZenStates' components: (a) the state machine name; (b) control buttons; (c) tasks associated with a state; (d) the initial state (in this case, also the one which is currently running, represented in green); (f) the blackboard; and transitions, composed by a guard condition (g) and its execution priority (e).

coordinates, key presses, various sensor inputs, incoming Open Sound Control (OSC²) messages, etc.

2) Making behaviors concrete: the Tasks

One challenge we needed to address concerns specifying concrete behaviors to happen inside the states. In the case of tools for developers, this behavior could be easily specified via programming language. But how could we make this easier for creative EUPs?

We solve the issue by introducing the notion of tasks: simple atomic behaviors representing actions (if it happens only once) or activities (if it keeps occurring over time) and that can be attached to states as off-the-shelf components [23].

Tasks work in the following way: Once a certain state is executed, all associated tasks are run in parallel. Each individual task also has its own set of idiosyncratic high-level parameters, that allow users to fine-tune its behaviors. In Figure 1, for instance, these high-level parameters are the intensity of the vibration, and the panning of a stereo audio-driven bass shaker. The UI controls to edit these parameters can be easily hidden/shown by clicking on the task, allowing users to focus their attention on the tasks they care the most.

Tasks are normally context-dependent, and need to be changed according to the domain of application. In creative interactive environments, for example, potential tasks could

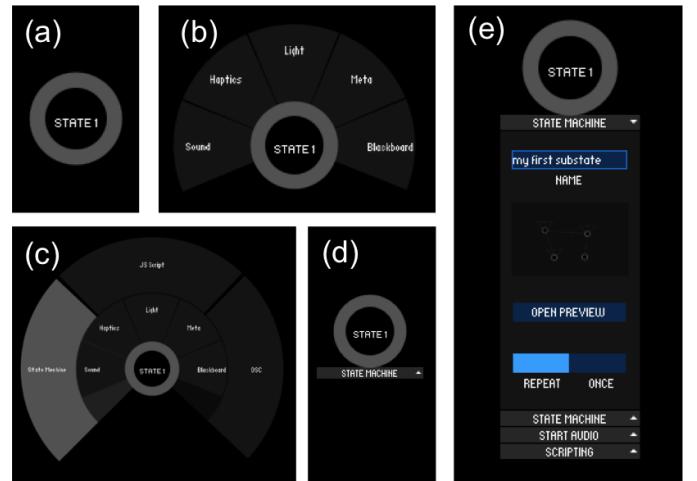


Fig. 3. Tasks inside states are reified with a contextual pie menu, as shown in the sequence above—(a), (b), (c), and (d). High-level parameters related to each task can be found by clicking on the task—as shown in (e).

be: sound-related tasks (e.g., start sound, control sound, stop sound), light-related tasks (start, control, and stop light), and haptics related tasks (start, control, and stop haptics)—as shown in Figure 3.

There are however two categories of tasks which are not context dependent: the blackboard-tasks, and the meta-tasks.

²<http://opensoundcontrol.org/>

Blackboard-tasks relate to creating new variables within the blackboard so that they can be later used anywhere inside the state machine. Our motivation is to increase reuse by providing users with a set of recurrent functionalities often found in interactive environments. Examples include oscillators, ramps, and random numbers.

Meta-tasks relate to extending ZenStates in situations where currently-supported functionalities are not enough. For example, OSC tasks allow communication with external media tools via the OSC protocol. JavaScript tasks allow custom JavaScript code to be incorporated to states. Finally, we have State Machine tasks, that allows nesting as shown in Figure 3.

3) Enriching transitions

We enrich transitions by incorporating two functionalities.

First, transitions make use of priorities that define the order they are going to be checked. This means that one state can have multiple transitions evaluated one after the other according to their priority (see Fig. 4). This allows users to write complex logical sentences similar to cascades of “if-then-else-if” in imperative languages. It also avoids potential logical incoherences raised by concurrent transitions.

Second, transitions support any Javascript expression as *guard conditions*, expressed as *transition and constraint events* as in [4]. In practice, this functionality combines logical operators (e.g. ‘&&’, ‘||’, ‘!’), mathematical expressions, and blackboard variables used either as events (e.g. ‘\$mousePressed’) or inside conditions (e.g. ‘\$mouseX > 0.5’). For instance, it is possible to set that if someone enters a room, the light should go off. Or if the mouse is pressed on a certain x and y coordinates, another page should be loaded.

4) Self-awareness

Self-awareness describes a set of meta characteristics belonging to states and tasks that are automatically added to the blackboard and become available to users. For example, states can have a status (e.g., is it running? Is it inactive?), sound tasks can have the current play position, the volume, and the audio pan as properties, etc. This feature can be useful in several ways. For example, we can set a transition to occur when all tasks within a certain state are completed (e.g. ‘\$State.Done’). Or yet, it is possible to have the volume of an audio file to increase as its playback position increases.

5) Live development & Reuse

Finally, ZenStates also borrows two additional features from [4] and [5]:

- **Live development:** Any element (e.g. tasks, transitions, and states) can be modified at runtime, allowing quicker process of experimentation and prototyping;
- **Reuse:** Previously defined interactive behaviors can easily be reused via nested state machines (that can be saved into files, exchanged, and then later imported). Here, we also expect to align with the principle of reuse as introduced by [26].

IV. IMPLEMENTATION

The design process of ZenStates has been iterative and user-centered with media artists. Steps included: a) *analysis*

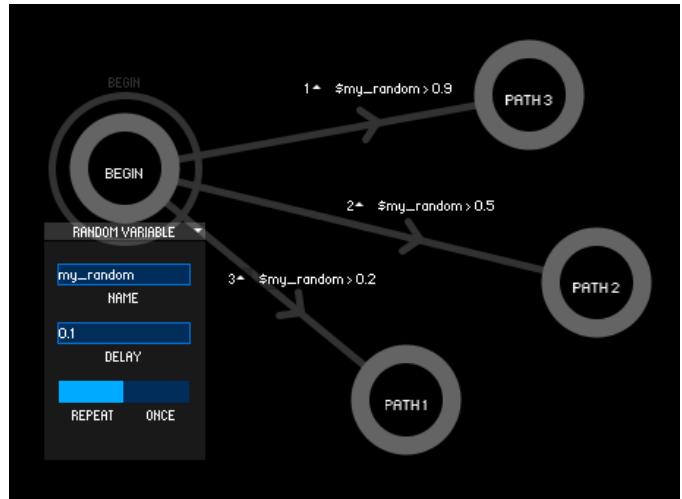


Fig. 4. Transitions have priorities attached to them. In this case, the blackboard variable ‘\$my_random’ generates a random number (between 0 and 1) which is then used to select a random path. The first transition to be checked is ‘\$my_random > 0.9’ because of its priority ‘1’. If false, transition with priority ‘2’ would then be checked. This process goes on until all transitions are checked. In case none is satisfied, the current state is executed once again before the next round of checking (repeat button is enabled). Transitions can be added, edited, and deleted via direct manipulation.

of existing alternatives; b) interviews with expert users; c) paper prototypes; d) development of scenarios to assess the potential of the tools in solving problems; e) observing the compositional process of two new media works; f) functional prototypes; and g) user-interface refinement over the course of think-aloud protocol sections with novice and expert users. While a full description of this process is beyond the scope of this paper, its direct result is the specification model as presented in this paper.

To validate this model, we implemented all features described here in a functional software prototype. In this prototype, abstract elements (namely the state machine, states, tasks, transitions, and the blackboard) are reified—according to the principle of reification introduced in [26]—to graphical objects that can be directly manipulated by the user [17]. These visual representations become the core of the user interaction and also allow users to infer the current state of the system at any time during execution: the current running state is painted green; states already executed are red; inactive states are gray (see Fig. 2). All images presented on this paper are actual screenshots of this functional prototype, also demonstrated in the supplementary video.

The prototype has been implemented in Java, using Processing³ as an external library for the graphics. The project is open-source and the source code is available online⁴. We stress that, as a prototype, this system is still under development and its usability needs improvement.

³<http://processing.org/>

⁴<https://github.com/jeraman/zenstates-paper-vlhcc2018>

V. EVALUATION

We argue ZenStates proposes an expressive (i.e., allow to develop a wide diversity of scenarios) yet easy-to-understand specification model for creative interactive environments. We have tested these claims in two steps. First, we have probed ZenStates' expressive power by designing 90 *exploratory scenarios*. Second, we have conducted a *user study* investigating ZenStates specification model in terms of its capacity to quickly and accurately describe interactive environments.

A. Exploratory scenarios

To explore the expressive power of ZenStates, we have developed 90 unique exploratory scenarios. These scenarios implement atomic audiovisual behaviors with different levels of complexity, based on a constrained set of inputs (either mouse, keyboard, or both), outputs (either background color of the screen, the sound of a sinewave, or both), and blackboard variables. The chosen behaviors are often used in the context of music/media arts. Examples include: Sinusoidal sound generators with user-controlled frequency (c.f., the project Reactable⁵) as implemented in the scenario '*mouse_click_mute_mousesexy_freq_amp*'; and contrasting slow movements and abrupt changes to create different moments in a piece (c.f., 'Test pattern' by Ryoji Ikeda⁶) implemented in the scenario '*random_flickering_random_wait_silence*'. The full list of exploratory scenarios—with implementation and video demonstration—is available as supplementary material⁷.

While small, these atomic scenarios can be further combined to one another via hierarchical state machines and transitions. Therefore, it is possible to create a potentially infinite number of new scenarios—much more complex than the atomic ones. This diversity illustrates the expressive power of ZenStates—especially considering the constrained set of input, outputs and blackboard variables that were used.

B. User study

We investigated if ZenState's specification model makes the development of interactive environments easier to understand. This model was compared to the specification model used by two popular interactive environments development tools: Puredata (hereafter, the *dataflow* model), and Processing (hereafter, the *structured* model).

1) Hypothesis: Our hypothesis is that Zenstates allows users to understand interactive environments specifications more *accurately*—that is, ZenStates would reduce the number of code misinterpretations by users—and *faster*—that is, ZenStates would reduce the amount of time necessary to understand the code—compared to the alternatives.

⁵https://www.youtube.com/watch?v=n1J3b0SY_JQ

⁶<https://www.youtube.com/watch?v=JfcN9Qhfir4>

⁷<https://github.com/jeraman/zenstates-paper-vlhcc2018/tree/master/scenarios>

2) Subjects and Materials: We recruited 12 participants (10 male, 1 female, and 1 not declared), aged 29 years old on average (*min* = 22, *max* = 46, *median* = 26, *SD* = 7.98). Participants were all creative EUPs with previous experience with interactive environments tools (e.g. media artists, composers, designers, and technologists), either professionals or students. Their expertise on computer programming ranged from novices to experts (*min* = 2 years, *max* = 23 years, *mean* = 7.67, *median* = 6, *SD* = 5.48). The most familiar languages for them were Python and Max/MSP (both with 4 people each), followed by Processing (2 people).

Regarding the experimental material, we have selected 26 of our exploratory scenarios representing a wide diversity of usage scenarios (i.e., either using blackboard variables; single input; and multimodal input). Each scenario was then specified using the three evaluated specification models (ZenStates, dataflow, and structured), resulting in 78 specifications. All these scenarios and their specifications are available online⁸.

3) Procedure: Our experimental procedure is based on [27], adapted and fine-tuned over a preliminary pilot with 6 participants. The resulting procedure is composed of 3 blocks—one block per specification model tested—of 6 trials.

In each block, participants were introduced to a specification model as presented in Figure 5. Participants were also given a small printed cheatsheet containing all possible inputs (i.e., mouse and keyboard), actuators (i.e., screen background color, and a sinewave), and language specific symbols that could appear over the trials. At this point, participants were allowed to ask questions for clarification, and were given some time to get used to the model.

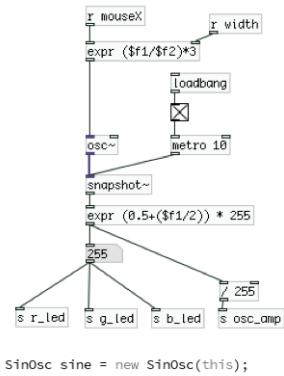
After this introduction, participants were introduced to one interactive environment specification—either ZenStates, dataflow, or a structured language—and to six different interactive environments videos showing real behaviors. Their task was *to choose which video they thought that most accurately corresponded to the specification presented*, as shown in Figure 6. Participants were instructed to be the most accurate, and to chose a video as quick as they could—without sacrificing accuracy. Only one answer was possible. Participants were allowed two practice trials and the cheatsheet could be consulted anytime.

Participants repeated this procedure for all three evaluated alternatives. Presentation order of the models was counterbalanced to compensate for potential learning effects. Similarly, the order of the six videos was randomized at each trial.

Our measured dependent variables were the *decision accuracy* (i.e., the percentage of correct answers), and the *decision time* (i.e., the time needed to complete the trial). As in [27], the decision time was computed as the trial total duration minus the time participants spent watching the videos. The whole procedure was automatized via a web application⁹.

⁸<https://github.com/jeraman/zenstates-paper-vlhcc2018/tree/master/data%20collection/html/scenarios>

⁹<https://github.com/jeraman/zenstates-paper-vlhcc2018/tree/master/data%20collection>



```

SinOsc sine = new SinOsc(this);

void draw() {
    float time = millis()/1000.0;
    float amplitude = 0.5;
    float frequency = ((float)mouseX/wid
    float value = 0.5;

    value = value + (float) (amplitude *
        Math.sin((2*Math.PI*frequency
    sine.amp(value));
    value = value*255;
    background(value, value, value);
}

```

Fig. 5. The three specification models presented to the participants: dataflow (top-left), structured (bottom-left), and ZenStates (right).

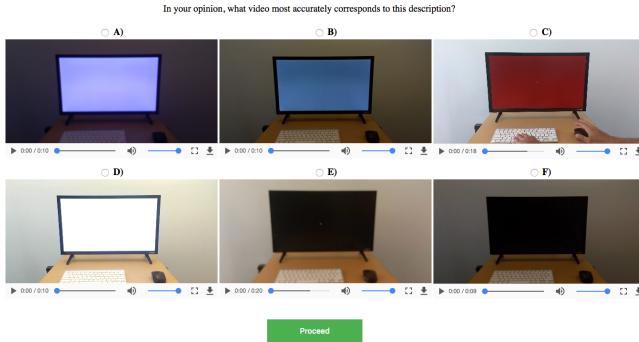


Fig. 6. Being introduced to one specification, participants needed to choose which one among six videos they thought that most accurately corresponded to the specification presented.

Finally, participants were asked about the specification models they had considered the easiest and the hardest to understand, followed by a short written justification.

4) Results: Data from HCI experiments has often been analyzed by applying null hypothesis significance testing (NHST) in the past. This form of analysis of experimental data is increasingly being criticized by statisticians [28], [29] and within the HCI community [30], [31]. Therefore, we report our analysis using estimation techniques with effect sizes¹⁰ and confidence intervals (i.e., not using p-values), as recommended by the APA [33].

Regarding *decision time*, there is a strong evidence for ZenStates as the fastest model (mean: 40.57s, 95% CI: [35.07,47.27]) compared to the dataflow (mean: 57.26s, 95% CI: [49.2,67.5]), and the structured model (mean: 70.52s,

¹⁰Effect size refers to the measured difference of means—we do not make use of standardized effect sizes which are not generally recommended [32].

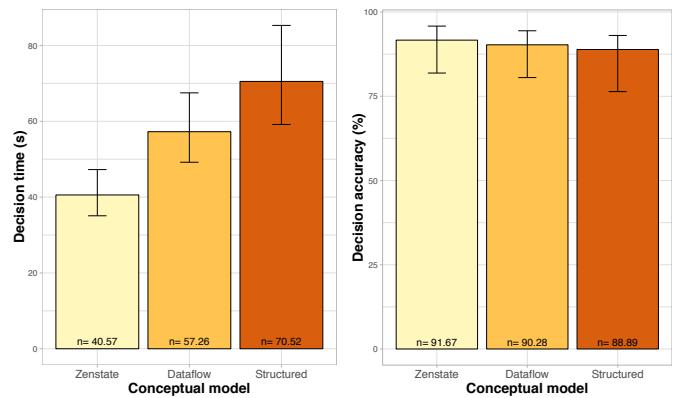


Fig. 7. *Decision time* (left) and *Accuracy* (right) per specification model. Error Bars: Bootstrap 95% CIs.

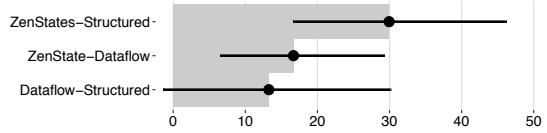


Fig. 8. *Decision time* pairwise differences. Error Bars: Bootstrap 95% CIs.

95% CI: [59.16,85.29]), as shown in Figure 7. We also computed pairwise differences between the models (Fig. 8) and their associated confidence intervals. Results confirm that participants were making their decision with ZenStates about 1.4 times faster than with the dataflow and 1.8 times faster than with the structured model. Since the confidence interval of the difference between dataflow and structured models overlaps 0, there is no evidence for differences between them.

Regarding *decision accuracy*, participants achieved 91.67% (95% CI: [81.89,95.83]) accuracy with ZenStates, 90.28% (95% CI: [80.55,94.44]) with the Dataflow model, and 88.89% (95% CI: [76.39,93.05]) with the structured model (see Fig. 7(right)). These results show quite high accuracy with all the specification models overall, although there is no evidence for one of them being more accurate than the other.

Finally, regarding the final questionnaire data, ZenStates was preferred by participants as the *easiest model to understand* (8 people), followed by the structured (3 people), and the dataflow (1 person) models. Participants written justifications reported aspects such as “*graphic display*”, “*code compartmentalization*”, and “*abstraction of low-level details*” as responsible for this preference. Similarly, dataflow (8 people) was considered the *hardest model to understand*, followed by structured (3 people), and ZenStates (1 person).

VI. LIMITATIONS

Our studies also revealed limitations in our specification model and its software interface, namely:

- **The blackboard:** Currently represented as a two-columns table on the top right of the screen, containing variable name and its value. While this initial approach

fulfills its goals (i.e., enhancing communication), it has limitations. A first limitation deals with representing a large amount of sensor data in the screen. For example, if a 3D depth camera is attached to the system (tracking x, y, and z points for all body joints), all detected joints would be added to the blackboard. For users interested in specific information (e.g. hand x position), the amount of visual data can be baffling. Another limitation concerns lack of support to high-level features (e.g. derivatives, and averages), and filtering, which are often as useful as raw sensor data. Further research is needed to improve the blackboard on these directions;

- **Physical environment alignment:** ZenStates assumes that the physical environment (i.e., sensor input, hardware for output media) and the tasks it supports (i.e., sound-related, light-related) is static and consistent, and that it would remain consistent along the execution. This assumption is reasonable when dealing with standard interaction techniques (e.g. WIMP tools for desktop, as in SwingStates [2]). However, because we are dealing with potentially more complex setups, it is possible that the assumption is no longer possible in certain cases. For example, in a certain artistic performance, some sensors might be added, or some light strobes removed during the performance. How to maintain this environment-software consistency in these dynamic cases? How should ZenStates react? These are open questions that need to be addressed in future developments;
- **Interface usability and stability:** The evaluation performed so far focuses only on readability of our specification model, not addressing ease of use or usability of the prototype interface that implements the model. We reason that ease of use and usability are not as relevant in such prototype stage as already-known problems would show up, limiting the evaluation of our specification model. At this stage, readability seems more effective as it could make specifications easier to understand, and potentially easier to learn, reuse, and extend. Nevertheless, we highlight that the usability of such interface would play a significant role in the effectiveness of our model. Examples to be improved include the obtuse expression syntax, using ‘\$’ to instantiate variables, and the small font size;

In addition to addressing these problems, we also plan to explore the usage of ZenStates in other creative contexts and to implement principles that could improve general support to creativity inside ZenStates (see [10] for examples).

VII. CONCLUSION

In this paper, we have analyzed the state of the art of development tools for programming rich interactive behaviors, investigating how these tools could support creative end-user programmers (e.g., media artists, designers, and composers) who typically struggle with technical development. As a solution, we introduced a simple yet powerful specification model

called ZenStates. ZenStates combines five key contributions–1) the blackboard, for communication; 2) the tasks, for concrete fine-tunable behaviors; 3) the prioritized guard-condition-based transitions; 4) self-awareness; and 5) live development & reuse–, exploring those in the specific context of interactive environments for music and media arts.

Our evaluation results suggest that ZenStates is expressive and yet easy to understand compared to two commonly used alternatives. We were able to probe ZenStates expressive power by the means of 90 exploratory scenarios typically found in music/media arts. At the same time, our user study revealed that ZenStates model was on average 1.4 times faster than dataflow model, 1.8 times faster than the structured model in terms of *decision time*, and had the highest *decision accuracy*. Such results were achieved despite the fact participants had no previous knowledge of ZenStates, whereas alternatives were familiar to participants. In the final questionnaire, 8 out of 12 participants judged ZenStates as the easiest alternative to understand.

We hope these contributions can help making the development of rich interactive environments—and of interactive behaviors more generally—more accessible to development-struggling creative communities.

ACKNOWLEDGMENT

The authors would like to thank Sofian Audry and Chris Salter for their support and numerous contributions, and everyone involved in the project ‘Qualified Self’ for their time.

REFERENCES

- [1] R. Blanch and M. Beaudouin-Lafon, “Programming rich interactions using the hierarchical state machine toolkit,” in *Proceedings of the working conference on Advanced visual interfaces - AVI ’06*. New York, New York, USA: ACM Press, 2006, p. 51.
- [2] C. Appert and M. Beaudouin-Lafon, “SwingStates: adding state machines to Java and the Swing toolkit,” *Software: Practice and Experience*, vol. 38, no. 11, pp. 1149–1182, sep 2008.
- [3] S. Oney, B. Myers, and J. Brandt, “ConstraintJS: Programming Interactive Behaviors for the Web by Integrating Constraints,” in *Proceedings of the 25th annual ACM symposium on User interface software and technology - UIST ’12*. New York, New York, USA: ACM Press, 2012, p. 229.
- [4] ———, “InterState: A Language and Environment for Expressing Interface Behavior,” in *Proceedings of the 27th annual ACM symposium on User interface software and technology - UIST ’14*. New York, New York, USA: ACM Press, 2014, pp. 263–272.
- [5] K. S.-P. Chang and B. A. Myers, “Creating interactive web data applications with spreadsheets,” in *Proceedings of the 27th annual ACM symposium on User interface software and technology - UIST ’14*. New York, New York, USA: ACM Press, 2014, pp. 87–96.
- [6] A. J. Ko, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, S. Wiedenbeck, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrence, and H. Lieberman, “The state of the art in end-user software engineering,” *ACM Computing Surveys*, vol. 43, no. 3, pp. 1–44, apr 2011.
- [7] F. Paternò and V. Wulf, Eds., *New Perspectives in End-User Development*. Cham: Springer International Publishing, 2017.
- [8] B. P. Bailey, J. A. Konstan, and J. Carlis, “Supporting Multimedia Designers: Towards more Effective Design Tools,” in *Proceedings of Multimedia Modeling*, 2001, pp. 267–286.
- [9] B. Myers, S. Y. Park, Y. Nakano, G. Mueller, and A. Ko, “How designers design and program interactive behaviors,” in *IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE, sep 2008, pp. 177–184.

- [10] B. Shneiderman, "Creativity support tools: accelerating discovery and innovation," *Communications of the ACM*, vol. 50, no. 12, pp. 20–32, 2007.
- [11] M. W. Krueger, "Responsive environments," in *Proceedings of the June 13-16, 1977, national computer conference on - AFIPS '77*. New York, New York, USA: ACM Press, 1977, p. 423.
- [12] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, jun 1987.
- [13] B. Ur, E. McManus, M. Pak Yong Ho, and M. L. Littman, "Practical trigger-action programming in the smart home," in *Proceedings of the 32nd annual ACM conference on Human factors in computing systems - CHI '14*. New York, New York, USA: ACM Press, 2014, pp. 803–812.
- [14] M. Puckette, "Max at Seventeen," *Computer Music Journal*, vol. 26, no. 4, pp. 31–43, dec 2002.
- [15] J. Noble, *Programming Interactivity: A Designer's Guide to Processing, Arduino, and openFrameworks*, 2nd ed. O'Reilly Media, 2012.
- [16] B. Victor, "Learnable programming," 2012. [Online]. Available: <http://worrydream.com/LearnableProgramming/>
- [17] B. Shneiderman, "Direct Manipulation: A Step Beyond Programming Languages," *Computer*, vol. 16, no. 8, pp. 57–69, aug 1983.
- [18] B. Hempel and R. Chugh, "Semi-Automated SVG Programming via Direct Manipulation," in *Proceedings of the 29th Annual Symposium on User Interface Software and Technology - UIST '16*. New York, New York, USA: ACM Press, 2016, pp. 379–390.
- [19] J. Jacobs, S. Gogia, R. Mch, and J. R. Brandt, "Supporting Expressive Procedural Art Creation through Direct Manipulation," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems - CHI '17*. New York, New York, USA: ACM Press, 2017, pp. 6330–6341.
- [20] J. F. Pane, C. A. Ratanamahatana, and B. A. Myers, "Studying the language and structure in non-programmers' solutions to programming problems," *International Journal of Human-Computer Studies*, vol. 54, no. 2, pp. 237–264, feb 2001.
- [21] F. Cuenca, K. Coninx, D. Vanacken, and K. Luyten, "Graphical Toolkits for Rapid Prototyping of Multimodal Systems: A Survey," *Interacting with Computers*, vol. 27, no. 4, pp. 470–488, jul 2015.
- [22] P. Dragicevic and J.-D. Fekete, "Support for input adaptability in the ICON toolkit," in *Proceedings of the 6th international conference on Multimodal interfaces - ICMI '04*. New York, New York, USA: ACM Press, 2004, p. 212.
- [23] J.-Y. L. Lawson, A.-A. Al-Akkad, J. Vanderdonckt, and B. Macq, "An open source workbench for prototyping multimodal interactions based on off-the-shelf heterogeneous components," in *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems - EICS '09*. New York, New York, USA: ACM Press, 2009, p. 245.
- [24] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Polit, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "STATEMATE: a working environment for the development of complex reactive systems," *IEEE Transactions on Software Engineering*, vol. 16, no. 4, pp. 403–414, apr 1990.
- [25] C. Appert, S. Huot, P. Dragicevic, and M. Beaudouin-Lafon, "Flow-States: Prototypage d'applications interactives avec des flots de données et des machines à états," in *Proceedings of the 21st International Conference on Association Francophone d'Interaction Homme-Machine - IHM '09*. New York, New York, USA: ACM Press, 2009, p. 119.
- [26] M. Beaudouin-Lafon and W. E. Mackay, "Reification, polymorphism and reuse," in *Proceedings of the working conference on Advanced visual interfaces - AVI '00*, no. May. New York, New York, USA: ACM Press, 2000, pp. 102–109.
- [27] K. Kin, B. Hartmann, T. DeRose, and M. Agrawala, "Proton++: A Customizable Declarative Multitouch Framework," in *Proceedings of the 25th annual ACM symposium on User interface software and technology - UIST '12*. New York, New York, USA: ACM Press, 2012, p. 477.
- [28] M. Baker, "Statisticians issue warning over misuse of P values," *Nature*, vol. 531, no. 7593, pp. 151–151, mar 2016.
- [29] G. Cumming, "The New Statistics," *Psychological Science*, vol. 25, no. 1, pp. 7–29, jan 2014.
- [30] P. Dragicevic, "Fair Statistical Communication in HCI," ser. Human-Computer Interaction Series, J. Robertson and M. Kaptein, Eds. Cham: Springer International Publishing, 2016, pp. 291–330.
- [31] P. Dragicevic, F. Chevalier, and S. Huot, "Running an HCI experiment in multiple parallel universes," in *Proceedings of the extended abstracts of the 32nd annual ACM conference on Human factors in computing systems - CHI EA '14*. New York, New York, USA: ACM Press, 2014, pp. 607–618.
- [32] T. Baguley, "Standardized or simple effect size: What should be reported?" *British Journal of Psychology*, vol. 100, no. 3, pp. 603–617, aug 2009.
- [33] G. R. VandenBos, *APA dictionary of psychology*. Washington, DC: American Psychological Association, 2007.

It's Like Python But: Towards Supporting Transfer of Programming Language Knowledge

Nischal Shrestha
NC State University
Raleigh, NC, USA
nshrest@ncsu.edu

Titus Barik
Microsoft
Redmond, WA, USA
titus.barik@microsoft.com

Chris Parnin
NC State University
Raleigh, NC, USA
cjparnin@ncsu.edu

Abstract—Expertise in programming traditionally assumes a binary novice-expert divide. Learning resources typically target programmers who are learning programming for the first time, or expert programmers for that language. An underrepresented, yet important group of programmers are those that are experienced in one programming language, but desire to author code in a different language. For this scenario, we postulate that an effective form of feedback is presented as a transfer from concepts in the first language to the second. Current programming environments do not support this form of feedback.

In this study, we apply the theory of learning transfer to teach a language that programmers are less familiar with—such as R—in terms of a programming language they already know—such as Python. We investigate learning transfer using a new tool called Transfer Tutor that presents explanations for R code in terms of the equivalent Python code. Our study found that participants leveraged learning transfer as a cognitive strategy, even when unprompted. Participants found Transfer Tutor to be useful across a number of affordances like stepping through and highlighting facts that may have been missed or misunderstood. However, participants were reluctant to accept facts without code execution or sometimes had difficulty reading explanations that are verbose or complex. These results provide guidance for future designs and research directions that can support learning transfer when learning new programming languages.

I. INTRODUCTION

Programmers are expected to be fluent in multiple programming languages. When a programmer switches to a new project or job, there is a ramp-up problem where they need to become proficient in a new language [1]. For example, if a programmer was proficient in Python, but needed to learn R, they would need to consult numerous learning resources such as documentation, code examples, and training lessons. Unfortunately, current learning resources typically do not take advantage of a programmer’s existing knowledge and instead present material as if they were a novice programmer [2]. This style of presentation does not support experienced programmers [3] who are already proficient in one or more languages and harms their ability to learn effectively and efficiently [4].

Furthermore, the new language may contain many inconsistencies and differences to previous languages which actively inhibit learning. For example, several blogs and books [5] have been written for those who have become frustrated or confused with the R programming language. In an online document [6], Smith lists numerous differences of R from

other high-level languages which can confuse programmers such as the following:

Sequence indexing is base-one. Accessing the zeroth element does not give an error but is never useful.

In this paper, we explore supporting learning of programming languages through the lens of *learning transfer*, which occurs when learning in one context either enhances (positive transfer) or undermines (negative transfer) a related performance in another context [7]. Past research has explored transfer of cognitive skills across programming tasks like comprehension, coding and debugging [8], [9], [10]. There has also been research exploring the various difficulties of learning new programming languages [11], [12] and identifying programming misconceptions held by novices [13]. However, limited research has focused on the difficulties of learning languages for experienced programmers and the interactions and tools necessary to support transfer.

To learn how to support transfer, we built a new training tool called Transfer Tutor that guides programmers through code snippets of two programming languages and highlights reusable concepts from a familiar language to learn a new language. Transfer Tutor also warns programmers about potential misconceptions carried over from the previous language [14].

We conducted a user study of Transfer Tutor with 20 participants from a graduate Computer Science course at North Carolina State University. A qualitative analysis on think-aloud protocols revealed that participants made use of learning transfer even without explicit guidance. According to the responses to a user satisfaction survey, participants found several features useful when learning R, such as making analogies to Python syntax and semantics. However, participants also pointed out that Transfer Tutor lacks code executability and brevity. Despite these limitations, we believe a learning transfer tool can be successful in supporting expert learning of programming languages, as well as other idioms within the same language. We discuss future applications of learning transfer in other software engineering contexts, such as assisting in code translation tasks and generating documentation for programming languages.

II. MOTIVATING EXAMPLE

Consider Trevor, a Python programmer who needs to switch to R for his new job as a data analyst. Trevor takes an online

```
1 df = pd.read_csv('Questions.csv')
2 df = df[df.Score > 0][0:5]
```

(a) Python

```
1 df <- read.csv('Questions.csv')
2 df <- df[df$Score > 0, ][1:5, ]
```

(b) R

Fig. 1. (a) Python code for reading data, filtering for positive scores and selecting 5 rows. (b) The equivalent code in R.

course on R, but quickly becomes frustrated as the course presents material as if he is a novice programmer and does not make use of his programming experience with Python and Pandas, a data analysis library. Now, Trevor finds himself ill-equipped to get started on his first task at his job, tidying data on popular questions retrieved from Stack Overflow (SO), a question-and-answer (Q&A) community [15]. Even though he is able to map some concepts over from Python, he experiences difficulty understanding the new syntax due to his Python habits and the inconsistencies of R. Trevor asks help from Julie, a seasoned R programmer, by asking her to review his R script (see Fig. 1) so he can learn proper R syntax and semantics.

Trevor’s task is to conduct a typical data analysis activity, tidying data. He is tasked with the following: 1) read in a comma-separated value (csv) file containing Stack Overflow questions 2) filter the data according to positive scores and 3) select the top five rows. Julie walks him through his Python code and explains how they relate to the equivalent code she wrote in R.

Julie teaches Trevor that R has several assignment operators that he can use to assign values to variables but tells him that the `<-` syntax is commonly used by the R community. However, she tells him that the `=` operator can also be used in R just like Python. To read a csv file, Julie instructs Trevor to use a built-in function called `read.csv()` which is quite similar to Python’s `read_csv()` function.

Moving on to the next line, Julie explains that selecting rows and columns in R is very similar to Python with some subtle differences. The first subtle difference that she points out is that when subsetting (selecting) rows or columns from a data frame in Python, using the `[` syntax selects rows. However, using the same operator in R will select columns. Julie explains that the equivalent effect of selecting rows works if a comma is inserted after the row selection and the right side of the comma is left empty (Figure 1b). Julie tells him that since the right side is for selecting columns, leaving it empty tells R to select all the columns. To reference a column of a data frame in R, Julie explains that it works almost the same way as in Python, except the `.` (dot) must be replaced with a `$` instead. Finally, Julie points out that R’s indexing is 1-based, so the range for selecting the five rows must start with 1, and unlike Python, the end index is inclusive. Trevor now has some basic knowledge of R. Could tools help Trevor in the same way Julie was able to?

III. TRANSFER TUTOR

A. Design Rationale

We created a new training tool called Transfer Tutor that takes the place of an expert like Julie and makes use of

learning transfer to teach a new programming language. Transfer Tutor teaches R syntax and semantics in terms of Python to help provide scaffolding [16] so programmers can start learning from a familiar context and reuse their existing knowledge. Our approach is to illustrate similarities and differences between code snippets in Python and R with the use of highlights on syntax elements and different types of explanations.

We designed Transfer Tutor as an interactive tool to promote “learnable programming” [17] so that users can focus on a single syntax element at a time and be able to step through the code snippets on their own pace. We made the following design decisions to teach data frame manipulations in R: 1) highlighting similarities between syntax elements in the two languages 2) explicit tutoring on potential misconceptions and 3) stepping through and highlighting elements incrementally.

B. Learning Transfer

Transfer Tutor supports learning transfer through these feedback mechanisms in the interface:

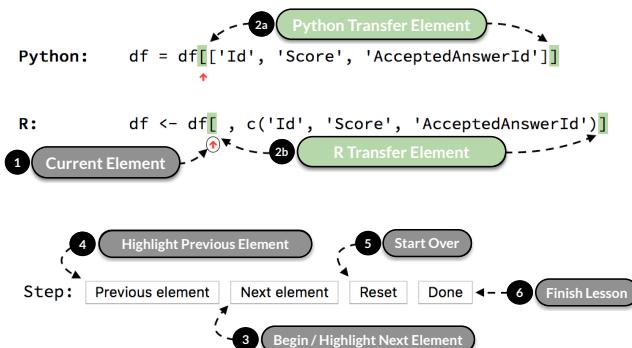
- **Negative Transfer:** ‘Gotchas’ warn programmers about a syntax or concept that either does not work in the new language or carries a different meaning and therefore should be avoided.
- **Positive Transfer:** ‘Transfer’ explanations describe a syntax or concept that maps over to the new language.
- **New Fact:** ‘New facts’ describe a syntax or concept that has little to no mapping to the previous language.

Each type of feedback consists of a highlighted portion of the code in the associated language (Python or R) with its respective explanation, which serves as affordances for transfer [18]. Furthermore, we support deliberate connections between elements, by allowing participants to step through the code, which helps them make a mindful abstraction of the concepts [19]. Finally, we focus on transferring declarative knowledge [20], such as syntax rules, rather than procedural knowledge, such as general problem-solving strategies.

C. User Experience

This section presents screenshots of Transfer Tutor and a use case scenario. The user experience of Transfer Tutor is presented from the perspective of Trevor who decides to use the tool to learn how to select columns of a data frame in R, a 2D rectangular data structure which is also used in Python/Pandas. The arrows and text labels are used to annotate the various features of the tool and are not actually presented to the users.

1) *Code Snippets and Highlighting*: Trevor opens up Transfer Tutor and notices that the tool displays two lines of code, where the top line is Python, the language that he is already familiar with and on the bottom is the language to learn which is R. Trevor examines the stepper buttons below the snippets and clicks ③ which begins the lesson and highlights some syntax elements:



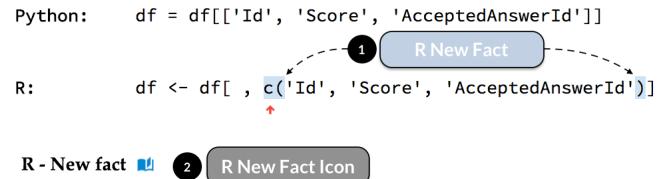
Travis notices ① points to the current syntax element in Python and R indicated by ②a and ②b. Trevor looks over to the right at the explanation box:



2) *Explanation Box*: Trevor sees ① which refers to a Python ‘transfer’ with ② showing the transfer icon. He reads ③ and learns that the [operator can be used in R. Transfer Tutor treats this syntax as a positive transfer since it can be reused. Trevor moves on to the next element:

Trevor looks at ① which is a red highlight on the Python code. He reads ② in the explanation box for clarification.

Trevor learns about a Python ‘gotcha’: the [[syntax from Python can’t be used in R. Trevor then reads ③ which explains an R ‘gotcha’ about how the [[syntax is legal in R, but semantically different from the Python syntax as it only selects a single column. In this case, Transfer Tutor warns him about a subtle difference, a negative transfer that could cause him issues in R. Trevor moves on to the next element and examines the elements that are highlighted blue:



In R, you can subset a data frame using vectors of the following types: character, positive integers, negative integers, and logical.

In this case, we create a character (string) vector to subset the data frame to select the columns (right side of comma). You can use the c() function to construct a vector, which is like a 1-dimensional array.

Trevor looks at ① then ② and realizes he’s looking at a ‘new fact’ about R. Transfer Tutor describes the c() function used to create a vector in R, which doesn’t have a direct mapping to a Python syntax.

3) *Code Output Box*: Finally, Trevor steps through the code to the end, and the code output box now appear at the bottom which displays the state of the data frame:

Trevor reads ① and inspects ② to understand the contents of the data frame in R and how it differs from Python’s data frame: 1) NaNs from Python are represented as NAs and 2) Row

indices start from 1 as opposed to 0. Transfer Tutor makes it clear that selecting columns of a data frame in R is similar to Python with some minor but important differences.

IV. METHODOLOGY

A. Research Questions

We investigated three research questions using Transfer Tutor to: 1) determine face validity of teaching a new language using an interactive tool 2) examine how programmers use Transfer Tutor and 3) determine which affordances they found to be useful for learning a new language.

RQ1: Are programmers learning R through Transfer Tutor? To identify if training through learning transfer is an effective approach in the context of programming, this question is used to determine the face validity of Transfer Tutor's ability to teach R.

RQ2: How do programmers use Transfer Tutor? Investigating how programmers use Transfer Tutor can identify when it supports learning transfer, and whether the affordances in the tool align with the way programmers reason about the problem.

RQ3: How satisfied are programmers with learning R when using the Transfer Tutor? We want to learn what features of Transfer Tutor programmers felt were useful to them. If programmers are satisfied with the tool and find it useful, it is more likely to be used.

B. Study Protocol

1) Participants: We recruited 20 participants from a graduate Computer Science course at our University, purposely sampling for participants with experience in Python, but not R. We chose to teach R for Python programmers because both languages are used for data science programming tasks, yet have subtle differences that are known to perplex novice R programmers with a background in Python [5], [6], [21].

Through an initial screening questionnaire, participants reported programming experience and demographics. Participants reported their experience with Python programming with a median of “1-3 years” (7), on a 4-point Likert-type item scale ranging from “Less than 6 months”, “1-3 years”, “3-5 years”, and “5 years or more” (1...). Participants reported a median of “Less than 6 months” (19) of experience with R programming (1...), and reported a medium of “1-3 years” with data analysis activities (1...). 16 participants reported their gender as male, and four as female; the average age of participants was 25 years ($sd = 5$).

All participants conducted the experiment in a controlled lab environment on campus, within a 1-hour time block. The first author of the paper conducted the study.

2) Onboarding: Participants consented before participating in the study. They were presented with a general instructions screen which described the format of the study and familiarized them with the interface. The participants then completed a pre-test consisting of seven multiple choice or multiple answer questions, to assess prior knowledge on R programming constructs for tasks relating to indexing, slicing,

and subsetting of data frames. The questions were drawn from our own expertise in the language and quizzes from an online text.¹ The presentation of questions was randomized to mitigate ordering effects. We also asked participants to think-aloud during the study, and recorded these think-aloud remarks as memos.

3) Study Materials: The authors designed four lessons on the topic of data frame manipulation, where each lesson consists of a one line code snippet in both languages and explanations associated with the relevant syntax elements. The authors also designed questions for the pre-test and post-test (see Table I). Finally, the authors designed a user satisfaction survey of Transfer Tutor. The study materials are available online.²

4) Tasks: Participants completed the following lessons on R: 1) assignment and reading data, 2) selecting columns, 3) filtering, and 4) selecting rows and sorting. Participants stepped through each lesson as described in Section III. Within each lesson, participants interacted with 5–8 highlights and corresponding explanation boxes.

5) Wrap-up: At the end of the study, participants completed a post-test containing the same questions as the pre-test. Participants completed a user satisfaction survey asking the participants for additional feedback on the tool. The survey asked them to rate statements about the usefulness of the tool using a 5-point Likert scale. These statements targeted different features of the tool such as whether or not highlighting syntax elements was useful for learning R. The survey also contained free-form questions for feedback regarding the tool such as the most positive and negative aspects, how they could benefit from using the tool and what features they would add to make it more useful. Finally, participants were given the opportunity to debrief for any general questions they may have had about the study.

C. Analysis

RQ1: Are programmers learning R through Transfer Tutor? We used differences in pre-test and post-test performance as a proxy measure for learning. We assigned equal weight to each question, with each question being marked as incorrect (0 points) or correct (1 point), allowing us to treat them as ordinal values. For the multiple answer questions, the participants received credit if they choose all the correct answers. A Wilcoxon signed-rank test between the participants' pre-test and post-test scores was computed to identify if the score differences were significant ($\alpha = 0.05$).

RQ2: How do programmers use Transfer Tutor? All authors of the paper jointly conducted an open card sort—a qualitative technique for discovering structure from an unsorted list of statements [22]. Our card sorting process consisted of two phases: *preparation* and *execution*. In the *preparation* phase, we extracted the think-aloud and observational data from the written memos into individual cards, with each card containing

¹<http://adv-r.had.co.nz>, chapters “Data Structures” and “Subsetting.”

²<https://github.com/alt-code/Research/tree/master/TransferTutor>

TABLE I
PRE-TEST AND POST-TEST QUESTIONS

ID	Question Text	Tot. ¹	Δ^2
1	Select all the valid ways of assigning a 1 to a variable ‘x’ in R.	0	18
2	Select all the valid vector types that can be used to subset a data frame.	13	2
3	How would one check if ‘x’ is the value NA?	0	20
4	Given a data frame df with column indices 1, 2, and 3, which one of these will cause an error?	10	3
5	Which one of these correctly selects the first row of a data frame df?	0	20
6	Which one of these correctly subsets the first five rows and the first column of a data frame df and returns the result as a data frame?	0	18
7	All of these statements correctly select the column ‘c’ from a data frame df <i>except</i>	0	1

¹Total number of participants who answered correctly in pre-test.

²Difference in the number of participants who answered correctly in pre-test and post-test.

a statement or participant observation. We labeled each of the cards as either being an indicator of positive transfer, negative transfer, or non-transfer. To do so, we used the following rubric to guide the labeling process:

- 1) Statements should not be labeled if it includes verbatim or very close reading of the text provided by Transfer Tutor.
- 2) The statement can be labeled as positive if it demonstrates the participant learning a syntax or concept from Python that can be used in R.
- 3) The statement can be labeled as negative if it demonstrates the participant learning a syntax or concept in R that is different from Python or breaks their expectation.
- 4) The statement can be labeled as a non-transfer if it demonstrates the participant encountering a new fact in R for which there is no connection to Python.

In the *execution* phase, we sorted the cards into meaningful themes. The card sort is open because the themes were not pre-defined before the sort. The result of a card sort is not to a ground truth, but rather, one of many possible organizations that help synthesize and explain how programmers interact with tool.

RQ3: How satisfied are programmers with learning R when using Transfer Tutor? We summarized the Likert responses for each of the statements in the user satisfaction survey using basic descriptive statistics. We also report on suggestions provided by participants in the free-form responses for questions, which include suggestions for future tool improvements.

V. RESULTS

In this section we present the results of the study, organized by research question.

A. RQ1: Are programmers learning R after using Transfer Tutor?

All participants had a positive increase in overall score ($n = 20$). The Wilcoxon signed rank test identified the post-test scores to be significantly higher than the pre-test scores ($S = 105$, $p < .0001$), and these differences are presented in Table I. Questions 1, 3, 5 and 6 provide strong support for learning transfer. In Question 2 and Question 4, most participants already supplied the correct answer with the pre-test; thus, there was a limited increase in learning transfer. The result of Question 7, however, was unexpected: no participants answered the pre-test question correctly, and there was essentially no learning transfer. We posit potential explanations for this in Limitations (Section VI). Based on these results, using test performance has face validity in demonstrating Transfer Tutor’s effectiveness in supporting learning transfer from Python to R.

B. RQ2: How do programmers use Transfer Tutor?

The card sorting results of the observational and think-aloud memos are presented in this section, organized into four findings.

Evidence of using transfer: We collected 398 utterances from our participants during their think-aloud during card sorting. All participants’ think-aloud contained utterances related to learning transfer. 35.9% of the total utterances related to transfer, revealing positive (18.9%) and negative transfers (66.4%). They also verbalized or showed behavior to indicate that they were encountering something that was new and didn’t map to something they already knew (14%). Other utterances not related to transfer involved verbatim reading of text or reflection on the task or tool.

Participants identified several positive transfers from Python, often without explicit guidance from Transfer Tutor. P4 guessed that the range for selecting a column in the Python code was equivalent to the one in R without Transfer Tutor explicitly mentioning this fact: “*both are the same, 2 colon in Python means 3 in R.*” Another participant correctly related Python’s dot notation to reference a data frame’s column to R’s use of dollar sign: “*Oh looks like \$ sign is like the dot.*” [P17]. This is evidence that programmers are naturally using learning transfer and Transfer Tutor helps support this strategy.

Participants also encountered several negative transfers from either Python or their previous languages. P15 thought the dot in the `read.csv()` function signified a method call and verbalized that the “*read has a csv function*” and later realized the mistake: “*read is not an Object here which I thought it was!*” P5 expressed the same negative transfer, thinking that “*R has a module called read.*”. This indicates a negative transfer from object-oriented languages where the dot notation is typically used for a method call.

Participants would also verbalize or show signs of behavior indicating that they have encountered a new fact, or a non-transfer, in R. This behavior occurred before progressing to the element with its associated explanation. P7 encountered the subsetting syntax in R and wondered, “*Why is the left side*

TABLE II
FOLLOW-UP SURVEY RESPONSES

	Likert Resp. Counts ¹						Distribution ²
	% Agree	SD	D	N	A	SA	
The highlighting feature was useful in learning about R.	95%	0	0	1	5	14	
Stepping through the syntax was useful in learning about R.	79%	0	1	3	2	14	
The explanations that related R back to another language like Python was useful.	89%	1	0	1	6	12	
The ‘new facts’ in the information box helped me learn new syntax and concepts.	95%	0	0	1	6	13	
The ‘gotchas’ in the information box were helpful in learning about potential pitfalls.	93%	0	2	0	6	12	
The code output box helped me understand new syntax in R.	79%	3	0	1	8	8	
The code output box helped me understand new concepts in R.	74%	2	0	3	7	8	

¹ Likert responses: Strongly Disagree (SD), Disagree (D), Neutral (N), Agree (A), Strongly Agree (SA).

² Net stacked distribution removes the Neutral option and shows the skew between positive (more useful) and negative (less useful) responses.

■ Strongly Disagree, ■ Disagree, ■ Agree; ■ Strongly Agree.

of the comma blank?” Another participant wondered about the meaning of a negative sign in front of R’s order function by expressing they “don’t get why the minus sign is there.” [P8].

Tool highlighted facts participants may have misunderstood or missed: The highlighting of the syntax elements and stepping through the code incrementally helped participants focus on the important parts of the code snippets. For additional feedback, one participant said “I was rarely confused by the descriptions, and the colorized highlighting helped me keep track of my thoughts and reference what exactly it was I was reading about with a specific example” [P17]. P13 had a similar feedback remarking that the “highlighting was good since most people just try to summarize the whole code at once.” However, a few participants found the stepper to progress the lesson too slowly. P17 read the entire line of code on the ‘Selecting rows and sorting’ lesson and said that they “didn’t understand drop=FALSE, hasn’t been mentioned” before Transfer Tutor had the opportunity to highlight it.

Reluctance of accepting facts without execution or examples: Participants were reluctant to accept certain facts without confirming for themselves through code execution, or without seeing additional examples. One participant was “not too sold on the explanation” [P2] for why parentheses aren’t required around conditions when subsetting data frames. Another participant expressed doubt and confusion when reading about an alternate [syntax that doesn’t require specifying both rows and columns: “Ok but then it says you can use an alternate syntax without using the comma” [P20]. Regarding the code output, one participant suggested that “it would’ve been more useful if I could change [the code] live and observe the output” [P18]. There were a few participants who wanted more examples. For example, P17 was unclear on how to use the [[syntax in R and suggested that “maybe if there was a specific example here for the [[that would help”.

Information overload: Although several participants reported that Transfer Tutor is “interactive and easy to use” [P13], there were a few who thought that there was “information overload in the textual explanations” [P1]. Some syntax

elements had lengthy explanations and one participant felt that “sometimes too many new things were introduced at once” [P18] and P5 expressed that “complex language is used” to describe a syntax or concept in R. Participants also expressed that they wanted “more visual examples” [P5].

C. RQ3: How satisfied are programmers with learning R when using Transfer Tutor?

Table II shows the distribution of responses for each statement from the user satisfaction survey, with each statement targeting a feature of Transfer Tutor. Overall, participants indicated that features of Transfer Tutor were useful in learning R. However, a few participants strongly disagreed about the usefulness of explanations relating back to R, and the output boxes. The free-form responses from participants offers additional insight into the Likert responses which will be discussed next.

The highlighting feature had no negative ratings and all participants indicated that it was useful to them in some way. One participant thought that “the highlighting drew [their] attention” [P2] while another commented that “it showed the differences visually and addressed almost all my queries” [P1].

The stepper received some neutral (3) ratings and one participant disagreed on its usefulness. Nevertheless, most participants did find the stepper useful and expressed that they “like how it focuses on things part by part” [P20].

Participants generally found the explanations relating R to Python was useful in learning R. One of the participants “liked the attempt to introduce R syntax based on Python syntax” [P18] and P14 thought that “comparing it with Python makes it even more easy to understand R language”. All participants thought this feature was useful except for one. This participant did not provide any feedback for why.

The ‘new facts’ explanations also had no negative ratings and was useful to all participants. Although participants didn’t speak explicitly about the feature, P8 expressed that there was “detailed explanation for each element” and P16 said that “Every aspect of the syntax changes has been explained very

well”. Most participants also found ‘gotchas’ to be useful. P7 for example said that “*Gotchas! were interesting to learn and to avoid errors while coding.*”

For the *explanation box*, some participants suggested that this affordance would need to “*reduce the need for scrolling and (sadly) reading*” [P2]. Still other participants wanted deeper explanations for some concepts, perhaps with “*links to more detailed explanations*” [P12]. For the *output boxes*, participants who disagreed with its usefulness suggested that the output boxes would be more useful if the output code be dynamically adjusted by changing the code [P6, P9, P12], and P17 suggested that the output boxes were “*a little difficult to read*” because of the small font.

VI. LIMITATIONS

A. Construct Validity

We used pre-test and post-test questions as a proxy to assess the participants’ understanding of R concepts as covered by Transfer Tutor. Because of time constraints in the study, we could only ask a limited number of questions. Consequently, these questions are only approximations of participants’ understanding. For instance, Question 7 illustrates several reasons why questions may be problematic for programmers. First, the question may be confusingly-worded, because of the use of *except* in the question statement. Second, the response may be correct, but incomplete—due to our scoring strategy, responses must be completely correct to receive credit. Third, questions are only approximations of the participants’ understanding. A comparative study is necessary to properly measure learning from using Transfer Tutor to other traditional methods of learning languages by measuring performance on programming tasks.

B. Internal Validity

Participants in the study overwhelmingly found the features of Transfer Tutor to be positive (Section V-C). It’s possible, however, that this positivity is artificially high due to social desirability bias [23]—a phenomenon in which participants tend to respond more positively in the presence of the experimenters than they would otherwise. Given the novelty of Transfer Tutor, it is likely that they assumed that the investigator was also the developer of the tool. Thus, we should be conservative about how we interpret user satisfaction with Transfer Tutor and its features.

A second threat to internal validity is that we expected Transfer Tutor to be used by experts in Python, and novices in R. Although all of our participants have limited knowledge with R, very few participants were also experts with Python or the Pandas library (Section IV). On one hand, this could suggest that learning transfer would be even more effective with expert Python/Pandas participants. On the other hand, this could also suggest that there is a confounding factor that explains the increase in learning that is not directly due to the tool. For instance, it may be that explanations in general are useful to participants, whether or not they are phrased in terms of transfer [24], [25].

C. External Validity

We recruited graduate students with varying knowledge of Python and R, so the results of the study may not generalize to other populations, such as industry experts. The choice of Python and R, despite some notable differences, are both primarily intended to be used as scripting languages. How effective language transfer can be when language differences are more drastic is still an open question; for example, consider if we had instead used R and Rust—languages with very different memory models and programming idioms.

VII. DESIGN IMPLICATIONS

This section presents the design implications of the results and future applications for learning transfer.

A. Affordances for supporting learning transfer

Stepping through each line incrementally with corresponding highlighting updates allows programmers to focus on the relevant syntax elements for source code. This helps novice programmers pinpoint misconceptions that could be easily overlooked otherwise, but prevents more advanced programmers from easily skipping explanations from Transfer Tutor. Despite the usefulness of always-on visualizations in nice environments [26], [27], an alternative implementation approach to always-on may be to interactively allow the programmer to activate explanations on-demand.

We found that live code execution is an important factor for programmers as they can test new syntax rules or confirm a concept. We envision future iterations of Transfer Tutor that could allow code execution and adapt explanations in the context of the programmers’ custom code.

Reducing the amount of text and allowing live code execution were two improvements suggested by the participants. This suggests that Transfer Tutor needs to reduce information overload and balance the volume of explanation against the amount of code to be explained. One solution is to externalize additional explanation to documentation outside of Transfer Tutor, such as web resources. Breaking up lessons into smaller segments could also reduce the amount of reading required.

B. Expert learning can benefit from learning transfer

To prevent negative consequences for experienced learners, we intentionally mitigated the expertise reversal effect [4] by presenting explanations in terms of language transfer—in the context of a language that the programmer is already an expert at. Participants in our study tried to guess positive transfers on their own, which could lead to negative transfers from their previous languages. This cognitive strategy is better supported by a tool like Transfer Tutor as it guides programmers on the correct positive transfers and warns them about potential negative transfers. We think that tools such as ours serve as a type of intervention design: like training wheels, programmers new to the language can use our tool to familiarize themselves with the language. As they become experts, they would reduce and eventually eliminate use of Transfer Tutor.

C. Learning transfer within programming languages

Our study explored learning transfer between programming languages, but learning transfer issues can be found within programming languages as well, due to different programming idioms within the same language. For example, in the R community, a collection of packages called tidyverse encourage an opinionated programming style that focuses on consistency and readability, through the use of a *fluent* design pattern. In contrast to ‘base’ R—which is usually structured as a sequence of data transformation instructions on data frames—the fluent pattern uses ‘verbs’ that pipe together to modify data frames.

D. Applications of learning transfer beyond tutorials

Learning transfer could be applied in other contexts, such as within code review tools, and within integrated development environments such as Eclipse and Visual Studio. For example, consider a scenario in which a software engineer needs to translate code from one programming language to another: this activity is an instance in which learning transfer is required. Tools could assist programmers by providing explanations in terms of their expert language through existing affordances in development environments. Learning transfer tools can be beneficial even when the language conversion is automatic. For example, SMOP (Small Matlab and Octave to Python compiler) is one example of a transpiler—the system takes in Matlab/Octave code and outputs Python code.³ The generated code could embed explanations of the translation that took place so that programmers can better understand why the translation occurred the way that it did.

Another potential avenue for supporting learning transfer with tools can be found in the domain of documentation generation for programming languages. Since static documentation can’t support all types of readers, authors make deliberate design choices to focus their documentation for certain audiences. For example, the canonical Rust book⁴ makes the assumption that programmers new to Rust have experience with some other language—though it tries not to assume any particular one. Automatically generating documentation for programmers tailored for prior expertise in a different language might be an interesting application for language transfer.

VIII. RELATED WORK

There are many studies on transfer between tools [28], [29], [30], [31] but fewer studies examining transfer in programming. Transfer of declarative knowledge between programming languages has been studied by Harvey and Anderson [20], which showed strong effects of transfer between Lisp and Prolog. Scholtz and Wiedenbeck [11] conducted a think-aloud protocol where programmers who were experienced in Pascal or C tried implementing code in Icon. They demonstrated that programmers could suffer from negative transfer of programming languages. Wu and Anderson conducted a similar study on problem-solving transfer, where programmers

who had experience in Lisp, Pascal and Prolog wrote solutions to programming problems [12]. The authors found positive transfer between the languages which could improve programmer productivity. Bower [32] used a new teaching approach called Continual And Explicit Comparison (CAEC) to teach Java to students who have knowledge of C++. They found that students benefited from the continual comparison of C++ concepts to Java. However, none of these studies investigated tool support.

Fix and Wiedenbeck [14] developed and evaluated a tool called ADAPT that teaches Ada to programmers who know Pascal and C. Their tool helps programmers avoid high level plans with negative transfer from Pascal and C, but is targeted at the planning level. Our tool teaches programmers about negative transfers from Python, emphasizing both syntax and semantic issues by highlighting differences between the syntax elements in the code snippets of the two languages. Transfer Tutor also covers pitfalls in R that doesn’t relate to Python.

We leverage existing techniques used in two interactive learning tools for programming, namely Python Tutor [33] and Tutorons [34]. Python Tutor is an interactive tool for computer science education which allows the visualization and execution of Python code. We borrowed the idea of Python Tutor’s ability to step through the code and pointing to the current line the program is executing to help the programmer stay focused. Head et al. designed a new technique of generating explanations or *Tutorons* that helps programmers learn about code snippets on the web browser by providing pop-ups with brief explanations of user highlighted code [34]. Although our tool does not automatically generate explanations for highlighted code, it uses the idea of providing details about syntax elements as the programmer steps through the syntax elements which are already highlighted for them.

IX. CONCLUSION

In this paper, we evaluated the effectiveness of using learning transfer through a training tool for expert Python developers who are new to R. We found that participants were able to learn basic concepts in R and they found Transfer Tutor to be useful in learning R across a number of affordances. Observations made in the think-aloud study revealed that Transfer Tutor highlighted facts that were easy to miss or misunderstand and participants were reluctant to accept certain facts without code execution. The results of this study suggest opportunities for incorporating learning transfer feedback in programming environments.

ACKNOWLEDGEMENTS

This material is based in part upon work supported by the National Science Foundation under Grant Nos. 1559593 and 1755762.

REFERENCES

- [1] S. E. Sim and R. C. Holt, “The ramp-up problem in software projects: A case study of how software immigrants naturalize,” in *International Conference on Software Engineering (ICSE)*, 1998, pp. 361–270.

³<https://github.com/victorlei/smop>

⁴<https://doc.rust-lang.org/book/second-edition/>

- [2] D. Loksa, A. J. Ko, W. Jernigan, A. Oleson, C. J. Mendez, and M. M. Burnett, “Programming, problem solving, and self-awareness: Effects of explicit guidance,” in *Human Factors in Computing Systems (CHI)*, 2016, pp. 1449–1461.
- [3] L. M. Berlin, “Beyond program understanding: A look at programming expertise in industry,” *Empirical Studies of Programmers (ESP)*, vol. 93, no. 744, pp. 6–25, 1993.
- [4] S. Kalyuga, P. Ayres, P. Chandler, and J. Sweller, “The expertise reversal effect,” *Educational Psychologist*, vol. 38, no. 1, pp. 23–31, 2003.
- [5] P. Burns. (2012) The R Inferno. [Online]. Available: <http://www.burns-stat.com/documents/books/the-r-inferno/>
- [6] T. Smith and K. Ushey, “aRrgh: a newcomer’s (angry) guide to R,” <http://arrgh.tim-smith.us>.
- [7] D. N. Perkins, G. Salomon, and P. Press, “Transfer of learning,” in *International Encyclopedia of Education*. Pergamon Press, 1992.
- [8] P. Pirolli and M. Recker, “Learning strategies and transfer in the domain of programming,” *Cognition and Instruction*, vol. 12, no. 3, pp. 235–275, 1994.
- [9] C. M. Kessler, “Transfer of programming skills in novice LISP learners,” Ph.D. dissertation, Carnegie Mellon University, 1988.
- [10] N. Pennington, R. Nicolich, and J. Rahm, “Transfer of training between cognitive subskills: Is knowledge use specific?” *Cognitive Psychology*, vol. 28, no. 2, pp. 175–224, 1995.
- [11] J. Scholtz and S. Wiedenbeck, “Learning second and subsequent programming languages: A problem of transfer,” *International Journal of Human-Computer Interaction*, vol. 2, no. 1, pp. 51–72, 1990.
- [12] Q. Wu and J. R. Anderson, “Problem-solving transfer among programming languages,” Carnegie Mellon University, Tech. Rep., 1990.
- [13] L. C. Kaczmarezyk, E. R. Petrick, J. P. East, and G. L. Herman, “Identifying student misconceptions of programming,” in *Computer Science Education (SIGCSE)*, 2010, pp. 107–111.
- [14] V. Fix and S. Wiedenbeck, “An intelligent tool to aid students in learning second and subsequent programming languages,” *Computers & Education*, vol. 27, no. 2, pp. 71 – 83, 1996.
- [15] “Stack Overflow,” <https://stackoverflow.com>.
- [16] R. K. Sawyer, *The Cambridge Handbook of the Learning Sciences*. Cambridge University Press, 2005.
- [17] B. Victor. (2012) Learnable programming. [Online]. Available: <http://worrydream.com/LearnableProgramming/>
- [18] J. G. Greeno, J. L. Moore, and D. R. Smith, “Transfer of situated learning,” in *Transfer on trial: Intelligence, cognition, and instruction*. Westport, CT, US: Ablex Publishing, 1993, pp. 99–167.
- [19] D. H. Schunk, *Learning Theories: An Educational Perspective*, 6th ed. Pearson, 2012.
- [20] L. Harvey and J. Anderson, “Transfer of declarative knowledge in complex information-processing domains,” *Human-Computer Interaction*, vol. 11, no. 1, pp. 69–96, 1996.
- [21] A. Ohri, *Python for R Users: A Data Science Approach*. John Wiley & Sons, 2017.
- [22] D. Spencer, *Card Sorting: Designing Usable Categories*. Rosenfeld, 2009.
- [23] N. Dell, V. Vaidyanathan, I. Medhi, E. Cutrell, and W. Thies, “‘Yours is better’: Participant response bias in HCI,” in *Human Factors in Computing Systems (CHI)*, 2012, pp. 1321–1330.
- [24] T. Kulesza, S. Stumpf, M. Burnett, S. Yang, I. Kwan, and W.-K. Wong, “Too much, too little, or just right? Ways explanations impact end users’ mental models,” in *Visual Languages and Human-Centric Computing (VL/HCC)*, 2013, pp. 3–10.
- [25] A. Bunt, M. Lount, and C. Lauzon, “Are explanations always important?” in *Intelligent User Interfaces (IUI)*, 2012, pp. 169–178.
- [26] H. Kang and P. J. Guo, “Omnicode: A novice-oriented live programming environment with always-on run-time value visualizations,” in *User Interface Software and Technology (UIST)*, 2017, pp. 737–745.
- [27] J. Hoffswell, A. Satyanarayan, and J. Heer, “Augmenting code with in situ visualizations to aid program understanding,” in *Human Factors in Computing Systems (CHI)*, 2018, pp. 532:1–532:12.
- [28] P. G. Polson, “A quantitative theory of human-computer interaction,” in *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*, 1987, pp. 184–235.
- [29] P. G. Polson, S. Bovair, and D. Kieras, “Transfer between text editors,” in *Human Factors in Computing Systems and Graphics Interface (CHI/GI)*, vol. 17, no. SI, 1986, pp. 27–32.
- [30] P. G. Polson, E. Muncher, and G. Engelbeck, “A test of a common elements theory of transfer,” in *Human Factors in Computing Systems (CHI)*, vol. 17, no. 4, 1986, pp. 78–83.
- [31] M. K. Singley and J. R. Anderson, “A keystroke analysis of learning and transfer in text editing,” *Human-Computer Interaction*, vol. 3, no. 3, pp. 223–274, 1987.
- [32] M. Bower and A. McIver, “Continual and explicit comparison to promote proactive facilitation during second computer language learning,” in *Innovation and Technology in Computer Science Education (ITICSE)*, 2011, pp. 218–222.
- [33] P. J. Guo, “Online Python Tutor: Embeddable web-based program visualization for cs education,” in *Computer Science Education (SIGCSE)*, 2013, pp. 579–584.
- [34] A. Head, C. Appachu, M. A. Hearst, and B. Hartmann, “Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code,” in *Visual Languages and Human-Centric Computing (VL/HCC)*, 2015, pp. 3–12.

Automatic Layout and Label Management for Compact UML Sequence Diagrams

Christoph Daniel Schulze

Department of Computer Science
 Christian-Albrechts-Universität zu Kiel
 Kiel, Germany
 Email: cds@informatik.uni-kiel.de

Gregor Hoops

Department of Computer Science
 Christian-Albrechts-Universität zu Kiel
 Kiel, Germany
 Email: email@gregorhoops.de

Reinhard von Hanxleden

Department of Computer Science
 Christian-Albrechts-Universität zu Kiel
 Kiel, Germany
 Email: rvh@informatik.uni-kiel.de

Abstract—Sequence diagrams belong to the most commonly used UML diagrams. There is research on desirable aesthetics, but to our knowledge no layout algorithms have been published. This might be due to the rigid specification of sequence diagrams that seems to make laying them out quite easy. However, as we argue here, naive algorithms do not always produce desirable solutions.

We present methods to produce compact layouts which we have implemented in a layout algorithm and evaluate them with 50 real-world sequence diagrams.

I. INTRODUCTION

UML's sequence diagrams [1, Section 17.8], such as the one in Fig. 1, specify *interactions* between *entities*. At their most basic, they consist of *lifelines* that each represent an entity and are connected by arrows which represent the exchange of *messages*. We will assume the reader to be familiar with the basics of sequence diagrams.

Sequence diagrams can grow rather large, which can decrease their usefulness. In this paper, we describe two ways to mitigate this problem.

A. Contributions

We will present and evaluate methods to reduce both the height and the width of sequence diagrams:

- *Vertical compaction*. As the number of messages in an interaction increases, sequence diagrams grow taller. We present vertical compaction as a means to decrease their height by allowing messages to share y coordinates.
- *Label management*. The number of lifelines affects a diagram's width, but so do the message labels. If we want to avoid impairing legibility due to crossings between lifelines and labels, these need to fit into the space between the lifelines, quite possibly pushing them apart. This problem is severe enough for companies to have developed guidelines for technical writers on how to draw sequence diagrams that fit into the available space [2]. We apply label management, first introduced in the context of another visual language [3], to sequence diagrams to improve this situation.

B. Related Work

As opposed to UML class diagrams, the layout of sequence diagrams has received comparatively little research attention.

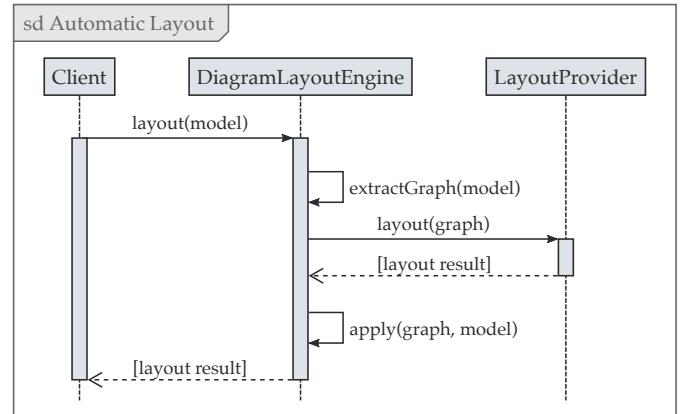


Fig. 1. A simple sequence diagram.

A paper by Wong and Sun [4] on desirable aesthetics of class and sequence diagrams is a case in point: while the authors reference four papers just on the layout of class diagrams, sequence diagrams are represented by only two papers [2], [5] which do not even describe layout algorithms, but merely general aesthetics. One reason for this situation might be that compared to class diagrams, sequence diagrams offer rather less freedom when it comes to their layout.

In fact, layout algorithms have been developed, but we are not aware of any having been published. Bennett et al. [6] have implemented a sequence diagram viewer as part of a tool used in a study. They did not, however, describe their layout algorithm. There are several sequence diagram editors available that turn specifications based on a domain-specific languages into diagrams. Examples are the *Quick Sequence Diagram Editor*,¹ *WebSequenceDiagrams*,² and *SequenceDiagram.org*.³ Again, details on their layout algorithms have not been published, but experiments showed that none apply any of the compaction techniques that we describe in Sec. II and Sec. III.

Poranen et al. [5] describe and partly formalize aesthetic criteria they believe to be desirable for the layout of sequence

¹<http://sdedit.sourceforge.net/>

²<https://www.websequencediagrams.com/>

³<https://sequencediagram.org/>

diagrams. Wong and Sun [4] pick up on those criteria and justify them with principles from perceptual theories. We will reference some of them throughout this paper.

Label management, first proposed by Fuhrmann [7], has since been successfully integrated into another visual language based on node-link diagrams [3]. Here, we will integrate it into sequence diagrams, which exhibit other characteristics and thus make for a valuable additional case study.

C. Outline

This paper is structured as follows. We start by discussing vertical compaction and how it can help reduce a diagram's height in Sec. II before turning to how label management can help reduce a diagram's width in Sec. III. After an evaluation in Sec. IV, we conclude in Sec. V.

An extended version of this paper that includes a more complete description of our layout algorithm is available as a technical report [8].

II. VERTICAL COMPACTION

Poranen et al. [5] consider a sequence diagram's size and aspect ratio to be mostly the result of its structure. In this point, we disagree.

Of course, the structure has an influence on the diagram's size, but there is room for vertical compaction. Usually, messages are laid out from top to bottom, each at a separate y coordinate, suggesting a total temporal ordering. However, only the order of messages at each lifeline is actually meaningful: if message a connects to a given lifeline above message b , a temporally occurs before b . Messages that are not related, either directly or indirectly, may well share y coordinates and thereby reduce their diagram's height.

In accordance with the *vertical distance* constraint defined by Poranen et al. [5], we divide the diagram into horizontal *communication lines*, a configurable amount of space apart, and restrict messages to run along these lines only. To allow messages to share y coordinates, we allow each communication line to host multiple messages.

Throughout the rest of this section, we will describe how we assign messages to communication lines and how we solve the problems that sharing communication lines can cause.

A. Assigning Messages to Communication Lines

To capture the order in which messages must appear in the diagram, we calculate an *element ordering graph* (see Fig. 2 for an example). Therein, each message is represented by a node and an edge runs from node a to node b if the following conditions are met:

- 1) There is a lifeline both messages connect to.
- 2) The message represented by a immediately precedes the message represented by b on that lifeline.

We call this an *element ordering constraint*, and it must be adhered to in the final layout by assigning messages to communication lines such that no two messages with direct or transitive ordering constraints end up on the same communication line. This problem is equivalent to the layer

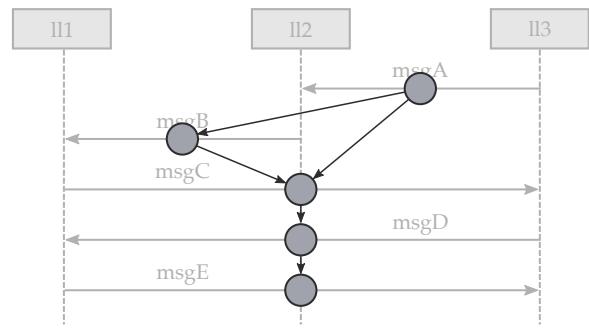


Fig. 2. A simple sequence diagram with its element ordering graph overlaid.

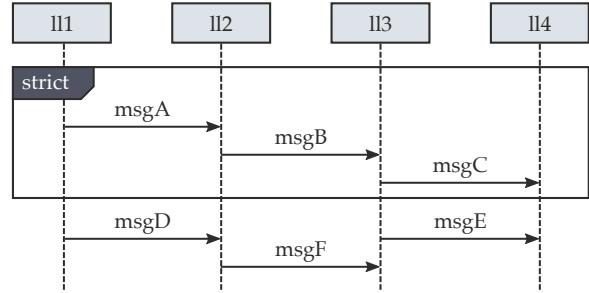


Fig. 3. Without further precautions, the ordering constraints placed on "msgD" would allow it to creep into the combined fragment "strict", sharing a communication line with "msgC". Adding additional element ordering constraints from "msgC" to "msgD" and "msgE" (which directly follow the fragment) solves the problem.

assignment problem which is at the core of the well-known *layered approach* to graph drawing by Sugiyama et al. [9]; our communication lines correspond to layers in the layer assignment problem. To solve it, we use the network simplex algorithm by Gansner et al. [10] to produce results that tend to keep messages close together on each lifeline.

B. Assignment Problems

Once *combined fragments* enter the picture, the approach starts to exhibit problems that need to be taken care of. Combined fragments, such as "strict" in Fig. 3, are rectangles drawn around a specific set of messages to combine them in some way. Unrelated messages must not enter the rectangle in order to not alter the diagram's semantics. There are two cases that can cause them to do so anyway.

For the first case, consider the sequence diagram in Fig. 3 again. Without further provisions the layout algorithm may end up moving "msgD" into the combined fragment, since the element ordering graph only indicates that it needs to be placed below "msgA" and "msgB". We can solve such cases by introducing additional ordering constraints between the bottommost messages in each fragment to those that follow messages in the fragment, but are not part of it themselves. In this case, we would add an edge from "msgC" to "msgD".

For the second case, consider the sequence diagram in Fig. 4a. Contrary to the first case, the messages on lifelines "ll2" and "ll3" have no relation at all to anything contained

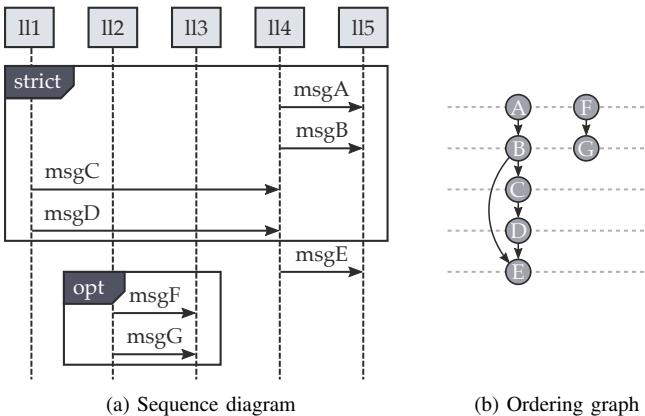


Fig. 4. Correct drawing of a sequence diagram with five lifelines, of which two (“ll2” and “ll3”) bear no relation to the others. Its element graph originally had both areas assigned to the same communication lines since there are no ordering constraints between the involved nodes.

in the “strict” combined fragment, causing us to not add additional element ordering constraints. Fig. 4b thus shows the assignment of the element ordering graph’s nodes to communication lines which would lead to an overlap of the two unrelated fragments.

Even if we had wanted to add additional ordering constraints, we would not have known which to add, exactly: should “msgF” and “msgG” be placed above or below the “strict” fragment? Such decisions can only be made once messages have been assigned to communication lines and possible overlaps can be detected. Note that rearranging the lifelines would solve the problem in this particular example, but the lifeline order may well have been fixed by the user or may have been computed to optimize for other goals.

We deal with this problem by post-processing the communication line assignment as follows. For each combined fragment, we compute the lifelines it spans by looking for the leftmost and for the rightmost lifeline incident to one of the fragment’s messages. The “strict” fragment in Fig. 4a, for example, spans all five lifelines whereas the “opt” fragment only spans two. We then iterate over all communication lines and keep a list of fragments for each lifeline that are currently open (or *active*) there. For each communication line, we perform four steps of computation.

1) *Step 1:* We iterate over the communication line’s nodes, looking for nodes that represent the start of a combined fragment. Let f_x be such a fragment. The fragment can begin at the current communication line if there is no other fragment f_y for which all of the following conditions are true:

- f_y has already been marked as being active.
- The sets of lifelines spanned by f_x and f_y have a non-empty intersection.
- f_x is not contained in f_y and f_y is not contained in f_x (otherwise it would be perfectly fine for them to overlap).

If we find no such fragment f_y , f_x can begin at the current communication line and is thus marked as being active at all lifelines it spans.

Taking the ordering graph from Fig. 4b as an example, we could start with either “msgA” or “msgF”. Let us suppose that we start with “msgA”, causing us to mark the “strict” combined fragment as active at all lifelines. For the fragment of “msgF”, we would detect a conflict with the “strict” fragment at lifelines “ll2” and “ll3” and thus refrain from activating it.

2) *Step 2:* We iterate over the current communication line’s nodes again, building an initially empty list of nodes that will have to be moved to the next communication line because they are in conflict with active fragments. If a node belongs to a fragment that has not been marked active by Step 1, it is added to our list. If a node represents a message that would cross an active fragment it is not part of, that too is added to our list.

In our example graph, “msgF” would be the only one added to the list while processing the first communication line.

3) *Step 3:* Each node in the list computed by the previous step is moved to the next communication line. If it has successors in the element ordering graph, that may invalidate the communication line assignment by placing two nodes with ordering constraints on the same communication line. We thus allow the movement to propagate through the following communication lines to restore the assignment’s validity.

In our example, we would move “msgF” to the second communication line. Since it now shares a communication line with “msgG”, we would move that to the third line.

4) *Step 4:* Finally, we look for active fragments that need to be deactivated. A fragment can cease to be active once we have encountered all of its messages.

In the example, the first time this happens is when we encounter “msgD” on communication line four. This is when we mark the “strict” fragment as not being active anymore, thereby allowing the “opt” fragment to become active once we process the next communication line.

III. LABEL MANAGEMENT

Too much text in a diagram has two effects: first, it increases the width of a diagram to a point where, if it is supposed to be drawn on screen in its entirety, it needs to be scaled down too much to be readable; and second, it puts much information on screen that may not actually be relevant to the viewer at a given moment. Label management [7], [3] solves this through *label management strategies* that take the original text and shorten or wrap it, optionally taking a desired *target width* into account. The latter can be provided by automatic layout algorithms since they know how long a label can be before it starts affecting the size of the diagram by pushing other elements apart.

Message labels of sequence diagrams are a good candidate for label management. If crossings between them and lifelines are to be avoided, they can push lifelines apart considerably and enlarge the diagram in the process.

We allow users to switch between two label management strategies. The first is a label management strategy which removes the arguments of method calls (*semantical abbreviation*). The second strategy simply cuts off the label text



Fig. 5. Height reduction achieved by vertical compaction for our set of sequence diagrams compared to without vertical compaction.

once it reaches the minimum amount of space available to it at the lifelines it is placed between (*syntactical abbreviation*), adding an ellipsis to indicate to users that more information are available. The available amount of space can be calculated by the layout algorithm by looking at the width of the involved lifelines, which is primarily determined by their title area.

Besides offering tool tips that provide access to the original text, we can also make good use of *focus and context* [11] in interactive viewing scenarios by only applying label management to those elements that are part of the context and displaying focussed elements in full detail. If the user selects a message, its label is focussed. If they select a lifeline, any incident messages are focussed.

IV. EVALUATION

We evaluated vertical compaction and our label management implementation with two aesthetics-based experiments. To do so, we used 50 sequence diagrams published on GitHub which we found among the first 5,000 items of a list of real-world UML models published by Helbig et al. [12].⁴ The sequence diagrams averaged 6.06 lifelines (for a total of 303) and 16.54 messages (827 total). The collected data and any scripts used to conduct the subsequent analysis are available online.⁵

A. Vertical Compaction

We wanted to answer the following research question: how much does vertical compaction reduce the height of sequence diagrams? To do so, every diagram was laid out twice: once with and once without vertical compaction. We measured the resulting height.

In our set of diagrams, 18% were affected by vertical compaction. Fig. 5 shows their change in height as compared to being laid out without vertical compaction (that is, without messages sharing y coordinates). If a diagram was affected by vertical compaction, its height was reduced by an average of about 47 pixels.

B. Label Management

We wanted to answer the following research questions: how much does vertical compaction reduce the width of sequence diagrams, and how does this influence the scaling factor with which we can display the diagram on a given drawing area?

Every diagram was laid out once with every available label management strategy as well as with label management switched off. We measured each diagram's width and height to compute the scaling factor increases.

⁴<http://oss.models-db.com/>

⁵<https://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/report-1804-data.zip>

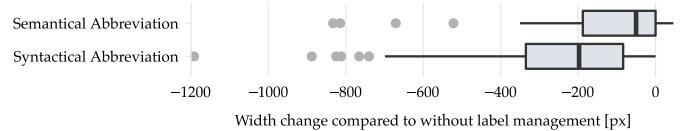


Fig. 6. Width reduction achieved by label management strategies four our set of sequence diagrams compared to disabling label management.

Fig. 6 shows how the width of our sequence diagrams changes subject to different label management strategies. Activating label management reduced the mean diagram width by up to almost 25%.

On average, the scaling factors increase to 1.03 the original scaling for semantical abbreviation and 1.04 for syntactical abbreviation.

C. Discussion

Vertical compaction never had a negative effect on the aesthetics of the diagrams in our test set. This leads us to believe that leaving it turned on will usually not be harmful, at least not regarding layout aesthetics.

When it comes to label management, the mean scaling factor increases where rather disappointing at first, but the values do make sense: quite often, sequence diagrams seem to be dominated by their height (true for 78% of diagrams in our data set), which dramatically reduces the impact of label management, a technique focussed on reducing the width of diagrams. If label management then makes narrow diagrams narrower, why use it in the first place? First, not all diagrams are dominated by their height. And second, applying label management allows lifelines to move closer together. This helps when zooming into a diagram since more of it can be displayed on screen at a time.

V. CONCLUSIONS AND OUTLOOK

In this paper we have discussed two ways to reduce the size of sequence diagrams: vertical compaction and label management. How effective the methods are depends on the actual sequence diagram and the application.

While we have evaluated the methods in terms of their influence on aesthetics, a user study may be interesting in order to establish how well they are received in practice.

We currently plan for an implementation of the algorithm we presented to be included in the Eclipse Layout Kernel (ELK) project, an open-source project that provides automatic layout algorithms and an infrastructure to support them.⁶ Until that has happened, preliminary versions can be made available upon request.

REFERENCES

- [1] Object Management Group, “OMG Unified Modeling Language Specification, Version 2.5.1,” Dec. 2017, <https://www.omg.org/spec/UML/2.5.1/>.

⁶<https://www.eclipse.org/elk/>

- [2] G. Bist, N. MacKinnon, and S. Murphy, "Sequence diagram presentation in technical documentation," in *Proceedings of the 22nd Annual International Conference on Design of Communication: The Engineering of Quality Documentation (SIGDOC '04)*. New York, NY, USA: ACM, 2004, pp. 128–133.
- [3] C. D. Schulze, Y. Lasch, and R. von Hanxleden, "Label management: Keeping complex diagrams usable," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '16)*, 2016, pp. 3–11.
- [4] K. Wong and D. Sun, "On evaluating the layout of UML diagrams for program comprehension," in *Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05)*, may 2005, pp. 317 – 326.
- [5] T. Poranen, E. Mäkinen, and J. Nummenmaa, "How to draw a sequence diagram," in *Proceedings of the Eighth Symposium on Programming Languages and Software Tools (SPLST'03)*, 2003.
- [6] C. Bennett, D. Myers, M. Storey, D. M. German, D. Ouellet, M. Salois, and P. Charland, "A survey and evaluation of tool features for understanding reverseengineered sequence diagrams," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 4, pp. 291–315, Jul. 2008.
- [7] H. Fuhrmann, "On the pragmatics of graphical modeling," Dissertation, Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, Kiel, 2011.
- [8] C. D. Schulze, G. Hoops, and R. von Hanxleden, "Automatic layout and label management for UML sequence diagrams," Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Technical Report 1804, Jul. 2018, ISSN 2192-6247.
- [9] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for visual understanding of hierarchical system structures," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 11, no. 2, pp. 109–125, Feb. 1981.
- [10] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo, "A technique for drawing directed graphs," *Software Engineering*, vol. 19, no. 3, pp. 214–230, 1993.
- [11] S. K. Card, J. Mackinlay, and B. Schneiderman, *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, Jan. 1999.
- [12] R. Hebig, T. H. Quang, M. R. V. Chaudron, G. Robles, and M. A. Fernandez, "The quest for open source projects that use UML: Mining GitHub," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS '16)*, 2016, pp. 173–183.

Evaluating the efficiency of using a search-based automated model merge technique

Ankica Barišić*, Csaba Debrezeni†‡, Daniel Varro†‡§, Vasco Amaral* and Miguel Goulão*

*NOVA LINCS, DI, FCT/UNL, Quinta da Torre 2829-516, Caparica, Portugal

†MTA-BME Lendület Cyber-Physical Systems Research Group, Budapest, Hungary

‡Budapest University of Technology and Economics, Budapest, Hungary

§McGill University, Montreal, Quebec, Canada

Abstract—Model-driven engineering relies on effective collaboration between different teams which introduces complex model management challenges. DSE Merge aims to efficiently merge model versions created by various collaborators using search-based exploration of solution candidates that represent conflict-free merged models guided by domain-specific knowledge.

In this paper, we report how we systematically evaluated the efficiency of the DSE Merge technique from the user point of view using a reactive experimental Software engineering approach. The empirical tests included the involvement of the intended end users (i.e. engineers), namely undergraduate students, which were expected to confirm the impact of design decisions. In particular, we asked users to merge the different versions of the same model using DSE Merge when compared to using Diff Merge. The experiment showed that to use DSE Merge participant required lower cognitive effort, and expressed their preference and satisfaction with it.

Index Terms—Domain-Specific Languages, Usability Evaluation, Software Language Engineering

I. INTRODUCTION

Model Driven Engineering (MDE) of critical cyber-physical systems (like in the avionics or automotive domain) is a collaborative effort involving heterogeneous teams which introduces significant challenges for efficient model management. While existing integrated development environments (IDEs) offer practical support for managing traditional software like source code, models as design artefacts in those tools are inherently more complex to manipulate than textual source code.

Industrial collaboration relies on version control systems (like Git or SVN) where differencing and merging artefacts is a frequent task for engineers. However, model difference and model merge turned out to be a difficult challenge due to the graph-like nature of models and the complexity of certain operations (e.g. hierarchy refactoring) that are common today.

In the paper, we focus on an open source tool developed within the MONDO European FP7 project [1] called DSE Merge [2]. DSE Merge presents a novel technique for search-based automated model merge [3] which builds on off-the-shelf tools for model comparison, but uses guided rule-based design space exploration (DSE) [4] for merging models. In general, rule-based DSE aims to search and identify various design candidates to fulfil specific structural and numeric constraints. The exploration starts with an initial model and systematically

traverses paths by applying operators. In this context, the results of model comparison will be the initial model, while target design candidates will represent the conflict-free merged model.

While existing model merge approaches detect conflicts statically in a preprocessing phase, this DSE technique carries out conflict detection dynamically, during exploration time as conflicting rule activations and constraint violations. Then multiple consistent resolutions of conflicts are presented to the domain experts. This technique allows incorporating domain-specific knowledge into the merge process with additional constraints, goals and operations to provide better solutions.

II. EVALUATION APPROACH

Practitioners are still experiencing problems to adopt modelling techniques in practice. Among other factors, developers seem to underestimate the importance of properly aligning the developed modelling tooling to support the techniques with the needs of their end users. We argue that this can only be done by properly assessing the impact of using the technique in a realistic context of use by its target domain users. Investment in the *usability evaluation* is justified by the reduction of development costs and increased revenues enabled by an improved effectiveness and efficiency [5].

Existing Experimental Software Engineering techniques [6] combined with Usability Engineering [7] can be adopted to support such evaluations [8]. This includes the application of experimental approaches, testing empirically with humans, and using systematic techniques to confirm the impact of design decisions on the usability of the developed tools.

Language usability can be defined as the degree to which a language can be used by specific users to meet their needs to achieve particular goals with effectiveness, efficiency and satisfaction in a particular context of use (adapted for the specific case of languages from [9]).

User-centered design (UCD) [10], [11] can contribute to more usable DSLs. For example, [12] presented an innovative visualisation environment, which eases and makes more effective the experimental evaluation process, implemented with the help of UCD. A visual query system was also designed and implemented following the UCD approach [13].

Conducting language usability evaluations is slowly being recognised as an essential step in the Language Engineering life-cycle [14]. An iterative approach allows us to trace usability

requirements and the impact of usability recommendations throughout the DSL development process [8].

III. EXPERIMENT

A. Experiment Preparation

The subjects with a different level of modelling expertise were selected to participate in experiment execution based on an online survey held before the experiment. Meanwhile, the development team prepared a demo for DSE Merge tool, the tasks and training material, and finally the virtual machine environment. The materials were evaluated during the pilot session that took place before the experiment execution. The participants of the pilot session were two academics that did not participate in the development of the evaluated tool.

Before starting the experiment, decisions have to be made concerning the *context of the experiment*, the *hypotheses under study*, the set of *independent and dependent variables* that will be used to evaluate the hypotheses, the *selection of subjects participating in the experiment*, the *experiment's design and instrumentation*, and also an *evaluation of the experiment's validity* [8]. The outcome of planning is the experimental evaluation design, which should encompass enough details to be independently replicable.

B. Experiment Objective

Our experiment addresses the following research question:

- *How usable is the proposed technique for performing the model merge operations when compared to the alternative?*

In particular we tested the following hypotheses regarding the use of DSE Merge when compared to the alternative:

Engineers can perform model merge operations ...

- *H1*: more effectively, producing correct results (i.e. merged models are of better quality).
- *H2*: more efficiently (i.e. obtained faster merged models).
- *H3*: more satisfactory (i.e. the modelling activity is perceived as more pleasant)
- *H4*: with less cognitive effort (i.e. lower modelling workload)

C. Experiment Context

The planning of the experiment started by defining explicitly the context of use for technology under evaluation, namely DSE Merge tool. The alternative, i.e. baseline support for model merge problem that is suitable for experimental comparison is identified to be the following:

- Diff Merge [15] shows all the changes to the user where the changes have to be applied manually one by one. Its strength is the user-friendly UI which is very intuitive for the novice users.
- EMF Compare [16] is default comparison and merge tool in the Eclipse environment. In each step, the tool shows only a subset of the changes that the user has to apply into the merge model. Its strength is the capability of handling very complex impacts of changes.

The alternative solutions are meant to support software engineers during the model merge process. The additional benefit claimed for the DSE Merge tool is its power to support domain experts in the same process without requiring from these experts a high level of programming expertise. DSE Merge is claimed to empower incorporation of domain-specific knowledge explicitly into the merge process. However, these two benefits can only be evaluated afterwards. This experiment was scoped to the similar context as alternative supports, to confirm its benefits in the familiar context described as follows:

- *User Profile* - target users for this experiment are expected to be software engineers
- *Technology* - all three tools are running over Eclipse IDE. OS during evaluation was Windows 7 on Desktop computer (Intel(R) Core(TM) i5 650@3.2GHz, 8 GB RAM, 19") or Lenovo Thinkpad T61p laptop (Intel T7700@2.4GHz, 4GB RAM, 15.4"). The two languages were tested per subject in the same machine.
- *Social and Physical environment* - the tool is expected to be used in a typical office environment, where the user is working individually by the desk using a laptop or desktop computer. Interaction is performed by use of the mouse, keyboard and the monitor.
- *Domain* - the domain chosen for the experiment was the Wind Turbine case study [17] developed by the industrial partners of the MONDO European Project, as it was previously well-defined and understood by our team.
- *Workflow* - due to the existence of the two different versions representing the same instance model, the user needs to find the best merge solution. The problem is more complex depending on the number of conflicts between the models. We defined the task (T0) as representative to problem reasoning based on domain example.

D. Experiment Flow

The experiment took place at the Budapest University of Technology and Economics [18]. The experimental process started by *Learning Session*, during which the subjects filled the *Background* questionnaire. After this they continue to solve the exercises during *Task Session* which was video recorded. Finally, during *Feedback Session* participants filled final questionnaire rating tools that they have used. Figure 1 depicts the flow of activities during the experiment, explicitly shows documents and treatments that were provided to participants, as well as the instruments that were used to collect the data.

1) *Training materials*: In the *Learning Session* the participants were allowed to ask questions and were provided with:

- the Wind Turbine Control System meta-model.
- the EMF-models demo video describing the use of Eclipse Modelling IDE and model merge problem.

During the *Tool Session* participants were not allowed to ask any question until the session was finished and were provided with following documents for each evaluated tool:

- the Demo video describing the use of the tool through a presentation of the task T0 that was defined in the experimental workflow context.

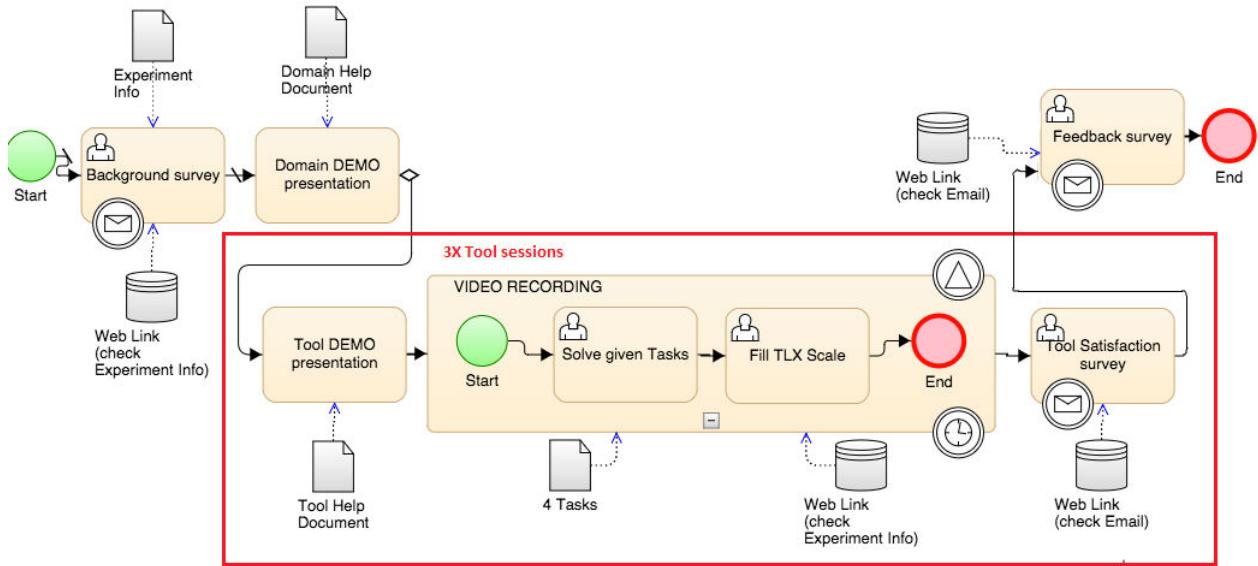


Fig. 1. Experiment treatments

- the Printed document containing explanations and screenshots presented in the demo video.

During the pilot session, the participants were asked to give the feedback about training directly on the printed materials. Time was estimated to be 10 minutes for *Learning Session*, while 5 minutes for each *Tool Sessions*.

2) *Experiment instruments and measurements*: These factors are presented in Table I.

TABLE I
INSTRUMENTS AND SCALES

	Instrument	Value
<i>Profile</i>	Availability Form, Background Questionnaire	[0-5]
<i>Duration</i>	Video recording	mm:ss
<i>Success</i>	Eclipse project delivery	[0-1]
<i>Cognitive Effort</i>	NASA TLX Scale	[0-1]
<i>Satisfaction</i>	Satisfaction Questionnaire	[(-1)-1]
<i>Preference</i>	Feedback Questionnaire	0 or 1

The data for calculating the *Profile* factor was collected through Availability and Background questionnaire. The *Profile* is influenced by experience in: Modelling; Education and programming; EMF Compare tool; Diff Merge tool; DSE Merge tool; and Wind Turbine metamodel. The *Profile* score (scaling from 0-5) was calculated as the average of all six Experience factors, to which it was added the value of 1 in a case that person had relevant Industry experience. In another case the person was assumed to be Academic.

Duration reflects the actual time taken to solve the tasks and was captured through video analysis.

Success reflects the multiplication of the Success Factor and the Quality Factor. The Quality Factor is defined for each task separately with the following values: 0, 0.25, 0.5, 0.75, 1. These predefined values reflect the number of conflicts that were resolved in contrast with a number of possible correct solutions delivered. The Success Factor took the following values: 1 (if the project reflects the set of correct solutions and

is delivered with success); 0.5 (project delivered but is not reflecting the set of correct solutions); 0 (no project delivery). The time to complete the 4 tasks was limited.

Cognitive Effort reflects the participant's workload during solving the task and is measured by a NASA TLX Scale [19].

The *Satisfaction* scale was reflecting average values regarding the following factors: Easy of Use; Confidence; Readability and Understandability of User Interface; Expressiveness; Suitability for complex problems; and Learnability.

The *Preference* factor reflects a clear preference toward one of the tools used based on a subset of Satisfaction criteria, that is annulled if in conflict with the same factor collected using Satisfaction Questionnaire.

All defined instruments were used during the pilot session. In an interview, the evaluator collected the suggestions and doubts regarding the surveys developed for the experiment.

3) *Tasks*: The representative tasks, of different level of complexity (see Table II), were defined and analysed to be used during experiment execution. During the pilot session, the cognitive effort for each task was estimated to be similar. Time was ranging between 3-5 minutes, while the success rate was high and it was a bit lower for more complex tasks.

TABLE II
TASK VALIDATION

Task	Model Size	Change Size	Solutions	Cognitive Effort	Time	Success
T1	Small	4	2	25.83	3:32	1
T2	Small	12	8	28.61	4:59	1
T3	Big	6	2	20.55	3:18	0.88
T4	Big	54	>million	24.02	4:27	0.83

Based on the obtained results and opinions of the participants during Pilot Session, Diff Merge was found to be a better alternative to DSE Merge for the designated tasks. Thus EMF Compare was excluded from the main experiment and left to be optional for the participants after solving the exercises using

TABLE III
COMPARING *Diff Merge* WITH *DSE Merge* - WELCH T TEST

	Diff Merge	DSE Merge	M Diff	S Err Diff	Lower CI	Upper CI	t	df	Sig. (2-tailed)	
H1	Success	0.82	0.90	-0.08	0.06	-0.20	0.03	-1.47	22.31	0.16
H2	Duration	1355.71	1289.36	66.36	188.90	-324.39	457.10	0.35	23.02	0.73
H3	Satisfaction	0.04	0.27	-0.23	0.09	-0.41	-0.05	-2.66	27.00	0.01
	- Frustration	58.00	51.43	6.57	9.68	-13.32	26.46	0.68	26.07	0.50
	- EasyToUse	0.00	0.50	-0.50	0.19	-0.90	-0.10	-2.58	25.63	0.02
	- Confidence	-0.03	0.32	-0.35	0.18	-0.73	0.02	-1.95	26.97	0.06
	- User Interface	0.07	0.21	-0.15	0.19	-0.54	0.25	-0.77	26.68	0.45
	- Expressiveness	0.20	0.57	-0.37	0.14	-0.66	-0.09	-2.68	26.42	0.01
	- Suitability	-0.13	0.32	-0.45	0.21	-0.88	-0.03	-2.19	26.62	0.04
	- Learnability	0.27	0.68	-0.41	0.18	-0.78	-0.05	-2.31	27.00	0.03
H4	TLX	65.31	53.09	12.22	5.93	0.03	24.41	2.06	26.03	0.05
	- Mental Demand	76.33	67.86	8.48	8.12	-8.46	25.42	1.04	19.97	0.31
	- Physical Demand	28.00	25.36	2.64	11.21	-20.35	25.64	0.24	26.99	0.82
	- Temporal Demand	46.67	51.07	-4.40	10.40	-25.79	16.98	-0.42	26.11	0.68
	- Performance	59.00	57.50	1.50	10.31	-19.68	22.68	0.15	26.30	0.89
	- Effort	66.67	58.21	8.45	7.94	-7.97	24.88	1.07	22.95	0.30

the evaluated tools. The experimental groups were divided into two (G1, G2). G1 received the first *Tool Session* for Diff Merge and then DSE Merge. G2 had the opposite sequence of G1.

IV. RESULTS

Subjects background - Out of 15 participants, 8 of them were from industry and 7 from academia. Most participants were experienced in programming and modelling, but none of them had experience in the Wind Turbine domain. Some participants had previous experience with alternative tools (mostly with EMF Compare), but only one had some basic knowledge of DSE Merge.

TABLE IV
SUBJECT BACKGROUND

	<i>Total</i>	<i>G1</i>	<i>G2</i>
<i>Number of participants</i>	15	6	9
<i>Profile</i>	1.65	1.92	1.39
<i>Industry</i>	56%	67%	44%

Comparative results - We compare the results for DSE Merge and Diff Merge in Table III. For each measured attribute, we present its mean value with Diff Merge, its mean value with DSE Merge, the mean difference between both, the standard error of that difference, the 95% confidence interval lower and upper boundaries, the Welch t-test statistic, its degrees of freedom and *p-value*. For hypothesis H1, although on average there was a slight improvement, we found no statistically significant difference between using both languages and, therefore, no evidence supporting the hypothesis that developers would achieve a higher success with DSE Merge. For hypothesis H2, although on average participants were slightly faster with DSE Merge, we found no statistically significant evidence supporting the hypothesis that the task would be performed more efficiently with DSE Merge when compared to Diff Merge. For H3, there was a statistically significant difference supporting the hypothesis that using DSE Merge leads to a higher satisfaction than using Diff Merge. This improvement was statistically significant concerning ease of use, confidence, expressiveness, suitability and learnability, with no significant difference concerning frustration or user

interface. Finally, concerning H4, overall, we found evidence supporting the hypothesis that the overall cognitive effort (NASA TLX global score) using DSE Merge was lower than using Diff merge. The difference is not attributable to any of the individual TLX scores. Finally, from the feedback questionnaire, we obtained the Preference factor of 11 for DSE Merge, while Diff Merge was only rated 1.

Threats to validity - Concerning the selection of the participants, they were all recruited in the same university. This creates a selection validity threat, as they may not fully represent the target population of DSE Merge. Besides, the sample size is relatively small. Replications of this evaluation should be independently conducted at other sites to mitigate these threats. Two other potential threats were hypothesis guessing where participants try to guess the hypotheses under study and change their behaviour as a result of it, and the experimenter's expectations. However, both the experimental evaluation and subsequent data analysis were conducted by researchers external to the development team of DSE Merge, thus mitigating both threats.

V. CONCLUSION

The results of the presented empirical study show that DSE Merge has clear advantages regarding the satisfaction (H3) of their users and the cognitive effort (H4) required to use it.

As future work, we plan to extend this study to subject modellers from the community of both practitioners and academics from outside the Budapest University. For that, we will make use of crowdsourcing platforms. This will allow us to improve both the statistical relevance of this study as well as to minimise the previously identified threat validity of the subjects representativity.

ACKNOWLEDGMENTS

The authors thank COST Action IC1404 Multi-Paradigm Modeling for Cyber-Physical Systems (MPM4CPS) H2020 Framework, as well as NOVA LINCS Research Laboratory (Grant: FCT/MCTES PEst UID/ CEC/04516/2013) and Project DSML4MA (Grant: FCT/MCTES TUBITAK/0008/2014).

REFERENCES

- [1] MONDO, “Scalable modelling and model management on the cloud,” project. [Online]. Available: www.mondo-project.org/, accessed: 07.11.2015
- [2] C. Debreceni, I. Ráth, D. Varró, X. D. Carlos, X. Mendialdua, and S. Trujillo, “Automated model merge by design space exploration,” in *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016*, ser. LNCS, vol. 9633. Springer, 2016, pp. 104–121.
- [3] M. Kessentini, W. Werda, P. Langer, and M. Wimmer, “Search-based model merging,” in *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. ACM, 2013, pp. 1453–1460.
- [4] A. Hegedus, A. Horváth, I. Ráth, and D. Varró, “A model-driven framework for guided design space exploration,” in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 173–182.
- [5] A. Marcus, “The ROI of usability,” in *Cost-Justifying Usability*, Bias and Mayhew, Eds. North-Holland: Elsevier, 2004.
- [6] V. R. Basili, “The role of controlled experiments in software engineering research,” in *Empirical Software Engineering Issues. Critical Assessment and Future Directions*, ser. LNCS, V. R. Basili, D. Rombach, K. Schneider, B. Kitchenham, D. Pfahl, and R. Selby, Eds. Springer Berlin / Heidelberg, 2007, pp. 33–37.
- [7] J. Nielson, *Usability Engineering*. AP Professional, 1993.
- [8] A. Barišić, V. Amaral, and M. Goulão, “Usability Driven DSL development with USE-ME,” *Computer Languages, Systems and Structures (ComLan)*, vol. ISBN 1477-, 2017.
- [9] International Standard Organization, “ISO/IEC FDIS 25010:2011 systems and software engineering – systems and software quality requirements and evaluation (SQuaRE) – system and software quality models,” March 2011. [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=35733
- [10] D. A. Norman and S. W. Draper, “User centered system design,” *Hillsdale, NJ*, 1986.
- [11] K. Vredenburg, J.-Y. Mao, P. W. Smith, and T. Carey, “A survey of user-centered design practice,” in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2002, pp. 471–478.
- [12] M. Angelini, N. Ferro, G. Santucci, and G. Silvello, “VIRTUE: A visual tool for information retrieval performance evaluation and failure analysis,” *Journal of Visual Languages & Computing*, vol. 25, no. 4, pp. 394–413, 2014.
- [13] E. Bauleo, S. Carnevale, T. Catarci, S. Kimani, M. Leva, and M. Mecella, “Design, realization and user evaluation of the SmartVortex Visual Query System for accessing data streams in industrial engineering applications,” *Journal of Visual Languages & Computing*, vol. 25, no. 5, pp. 577–601, 2014.
- [14] T. Kosar, M. Mernik, and J. Carver, “Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments,” *Empirical Software Engineering*, vol. 17, no. 3, pp. 276–304, 2012.
- [15] “EMF Diff/Merge,” https://wiki.eclipse.org/EMF_DiffMerge, accessed: 2018-07-25.
- [16] “EMF compare,” <https://www.eclipse.org/emf/compare/>, accessed: 2018-07-25.
- [17] A. Gómez, X. Mendialdua, G. Bergmann, J. Cabot, C. Debreceni, A. Garmendia, D. S. Kolovos, J. de Lara, and S. Trujillo, “On the opportunities of scalable modeling technologies: An experience report on wind turbines control applications development,” in *Modelling Foundations and Applications - 13th European Conference, ECMFA 2017*, ser. LNCS, vol. 10376. Springer, 2017, pp. 300–315.
- [18] A. Barišić, “STSM Report: Evaluating the efficiency in use of search-based automated model merge technique,” in *Multi-Paradigm Modelling for Cyber-Physical Systems (MPM4CPS)*, no. COST action IC1404. European cooperation in science and technology, 2016. [Online]. Available: http://mpm4cps.eu/STSM/reports/material/STSM_report-Ankica_Barisic.pdf
- [19] S. G. Hart and L. E. Staveland, “Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research,” *Advances in psychology*, vol. 52, pp. 139–183, 1988.

SiMoNa: A Proof-of-concept Domain Specific Modeling Language for IoT Infographics

Cleber Matos de Moraes

Universidade Federal da Paraíba

João Pessoa/PB, Brazil

Email: cmorais@gmail.com

Judith Kelner, Djamel Sadok

Universidade Federal de Pernambuco

Recife/PE, Brazil

Email: {jk,jamel}@gprt.ufpe.br,

Theo Lynn

Dublin City University

Dublin, Ireland

Email: theo.lynn@dcu.ie

Abstract—The Internet of Things (IoT) has emerged as one of the prominent concepts in academic discourse in recent times reflecting a wider trend by industry to connect physical objects to the Internet and to each other. The IoT is already generating an unprecedented volume of data in greater varieties and higher velocities. Making sense of such data is an emerging and significant challenge. Infographics are visual representations that provide a visual space for end users to compare and analyze data, information, and knowledge in a more efficient form than traditional forms. The nature of IoT requires a continuum modification in how end users see information to achieve such efficiency gains. Conceptualizing and implementing Infographics in an IoT system can thus require significant planning and development for both data scientists, graphic designers and developers resulting in both costs in terms of time and effort. To address this problem, this paper presents SiMoNa, a domain-specific modeling language (DSML) to create, connect, interact, and build interactive infographic presentations for IoT systems efficiently based on the model-driven development (MDD) paradigm.

Index Terms—Domain Specific Modeling Language, Model Driven Development, Internet of Things, Data Visualization, Infographics,

I. INTRODUCTION

The Internet of Things (IoT) has emerged as one of the prominent concepts in academic discourse in recent times reflecting a wider trend by industry to connect physical objects to the Internet and to each other. The IoT is already generating an unprecedented volume of data in greater varieties and higher velocities [1] [2] [3]. An IoT system spectrum deals with many variables that can be used characterize each application [4].

For example, one can characterise an IoT application with just two variables, such as area and data intensity. A Smart Home would be use case scenario with a small area and a low data intensity whereas a Smart City could be a use case with a large area and high data intensity. Similarly, a Smart Factory may be characterised in terms of a small area, for example a warehouse, with high data intensity resulting from the use of sensors. While all these applications fall within the Internet of Things, each one not only has a different type of area and data intensity but criticality.

In each of the above scenarios, the data volume is large. The Smart City and Smart Factory also require that data processing

and use are near-real time. Despite this, most of the data collected by an IoT system is not processed [5] or often isn't even stored for future analysis. In fact, it is estimated that less than 1% of IoT data is used for decision making. Using this deluge of data to build information and knowledge for decision making is a significant business challenge. Machine assistance using machine learning techniques are enhancing this ability [6], generating new information to feed and help decision making [7] but ultimately the human decision maker plays a central role. The human eye is the most data intensive and efficient sense in the human body [8] playing a role facilitating memorization in many cases.

The nature of IoT requires a continuum modification in how end users see information to achieve such efficiency gains. Conceptualizing and implementing Infographics in an IoT system can thus require significant planning and development for both data scientists, graphic designers and developers resulting in costs both in terms of time and effort. To address this problem, this paper presents SiMoNa, a domain-specific modeling language (DSML) to create, connect, interact, and build interactive Infographic presentations for IoT systems efficiently based on the model-driven development (MDD) paradigm.

The language proposed has its roots in prior IoT Domain-specific Languages, such as [9], [10], but SiMoNa is more focused on Infographic visualization rather than the IoT architecture as a whole. From a visual perspective, [11] deals with the representation of Big Data in a geo-spatial context. From a domain modeling language perspective, the works [12], [13] are very similar to SiMoNa, but applied to a different domain i.e. automated software engineering tools.

This paper is organized as follows. Section II introduces Infographics and data visualization as human interfaces to information. It also introduces Model-driven Development (MDD) and its correlated Domain Specific Modeling Languages (DSML) as a strategic methodology to address the infographic dynamics in IoT systems. Next, the SiMoNa DSML is presented with its meta-model and elements, followed by the Conclusion and Future Works(section IV), and References.

A. Contributions

The main contributions of this work are:

This work is partly funded by the Irish Centre for Cloud Computing and Commerce (IC4), a Enterprise Ireland/IDA Technology Centre.

- Presentation the Infographic perspective as tool to address the IoT visualization;
- Applies modeling, creation and implementation of a DSML for Infographics in a replicable environment;
- Propose a proof-of-concept solution for Infographic interfaces using the Model-Driven Development paradigm.

II. INFOGRAPHICS AND DATA VISUALIZATION

Graphics reveal data. With this premise, the visual display of quantitative information [14] inspires designers and statisticians to create accurate visual representations of such data. In the same way, computer scientists are exploring the opportunities raised through the intersection of digital and interactive graphics and Big Data. Computer science makes a significant contribution to data visualization through reducing the economics of creating the graphic, increasing flexibility to recreate a graphic, and enhancing user interaction with the graphic.

The cost of handling and interpreting additional information in up-to-date digital environments are extremely low for most graphics. When combined with interactivity, the simultaneous observation and interaction with a graphic creates a cognitive dual visual experience for the user [15] [16]. This interactive experience triggers two different parts of the brain simultaneously. Firstly, the part of the brain controlling visual conscious perception (vision-for-perception) is activated. Secondly, call-for-action visual perception is activated (vision-for-action). For example, when someone sees a cup of coffee on the table, this has been processed in two parts of the brain simultaneously. First, the image is separated from the background so that the cup is perceived within the environment stimulus. Second, the call-for-process is instigated to map the physical motor system to trace and pick up the cup. Even if the person does not want to pick up the cup, the brain prepares the human motor system to be ready to do so. In such a way, the user experiences both a visual stimulus and a physical call-for-action when interacting with a touchscreen panel. This dual mental process is especially required in a high skilled task use scenarios. IoT systems are often such scenarios. Thus, the ability for an IoT system to communicate data visually and interactively is critical for an IoT system, as the end user can perceive a event and act accordingly in a complex environment [16].

A. From Graphics to Infographics

Infographics are a diagrammatic representations of data [17]. Infographics are more complex than a series of graphics presented together. At its core, an Infographic represents a purposeful diagramming of each information source, thus each graphic (and even non-graphic information) have a predefined purpose (and associated meaning) in the visual space. There is a narrative in the Infographic scope, with syntax and semantics.

The three main elements of Infographics are data substance, relevant statistics, and design [14]. In the IoT context, the data is often provided by sensors. Despite this, not all sensor data is

relevant for a specific use case. For example, even if the power distribution unit (PDU) could offer the wattless [18] charge information, this is not necessarily useful for energy efficiency decision making in a Smart Home use case. In contrast, that information might be critical in a industrial or business use case. Useless data would represent noise [19] to visualization. The data substance must fit the use case, regarding both the quantity and quality of data to be presented to the user.

Statistics are at the core of data processing. Merely presenting data on a screen does not help the end user in the decision making process. The system must offer information in a clear and thoughtful way to enhance the data-information-knowledge continuum [20]. The capability to process data, compare it, and present those results to the user in a meaningful way is both central and critical to utility of Infographics.

Design is the final presentation of all the information to the user. A narrative bonds the data presentation scope to facilitate the user's perception of information [21]. This narrative is composed by the aesthetic applied in a effective way to present information. As a language grammar, the visual representation has presentation rules [22].

Based on those these three pillars - data substance, relevant statistics, and design - this work presents some basic principles to define interactive Infographic systems: label=(0)

- 1) The data source must fit the user requirements in terms of relevance, quantity, and timeliness;
- 2) The Infographic must allow the user to compare precisely the data presented in the same context;
- 3) The design narrative must be consistent and have a meaning for each section of a Infographic;
- 4) The Infographic should allow the user to query, investigate, explore, mark, create triggers, and compare data and information in the same interface;
- 5) The Infographic system should react to a data level defined by the user and automatically store new information while feeding back this new information for visualization and analysis.

Conceptualizing and implementing Infographics in an IoT system can thus require significant planning and development for data scientists, graphic designers and software developers. Each new element in this complex representation system (new data sources, new graphics, new statistical methods or new narratives) incurs costs in terms of time and effort. To address this problem, this work considers the use of Model-driven Development (MDD) as a key strategy to deal with this complexity.

B. Model-driven Development (MDD)

Model-driven Development (MDD) is an evolution of the software diagrams and software development methodologies. According to [23], instead of requiring developers to spell out every detail of a system's implementation using a programming language, creating documentation and code, it would be more efficient if developers could just model the system, describing the architecture and functionality.

In this way, by using MDD, developers can deal with high level abstractions to define their system requirements, and then automatically generate the required code [24]. The code samples for the code generator are provided by the domain specialists and tested in the unit of production. As a consequence, the software development becomes more resilient to requirement changes (especially in dynamic scenarios, such as IoT systems) and the generated code has higher quality.

To make use of MDD, it is necessary to define a Domain Specific Modeling Language (DSML) to describe the system requirements. DSML are easier to specify, understand and maintain. According to [25], DSML promotes productivity of modeling and also contributes to model quality since the DSML concepts should be the result of an especially thorough development process. The integrity of models is achieved because the syntax and semantics of a DSML can prevent nonsensical models. Furthermore, *a DSML will often feature a special graphical notation (concrete syntax) that helps to improve the clarity and comprehensibility of models* [25].

III. SiMoNa, AN INFOGRAPHIC DSML

This work presents SiMoNa, an Infographic Domain-Specific Modeling Language. SiMoNa is an acronym for Monitoring and Analytics Information System in Portuguese. It is an extension of the SiMoN IoT system, developed by [26]. The main requirement of the language is to address a wide range of IoT Infographic in a quick and efficient way through the MDD paradigm.

A. Tool and Meta-metalinguage

SiMoNa was build with The MetaEdit+ Workbench 5.5 software and its meta-metamodeling language [27]. The MetaEdit+ uses a GOPPRR meta-modeling language, widely used in software development and research. GOPPRR is an

acronym for the language's base types: Graph, Object, Port, Property, Relationship and Role.

The main reason to select MetaEdit+ relies on the experience, replicability and extensibility of the internal process of its metamodel. In the proof-of-concept level of the language, it allowed fast try and iterate circles. Anyone can validate and further extend this work as needed. Also, the main approach to the model is visual so a graphical language has better visual representation of the Infographics displacement and configuration.

B. Meta-model

The meta-model elements are presented in Figure 1. The main part of the meta-model is the Infographic. In its first version, each Infographic screen is composed from up to 5 graphics and a panel for notices and warnings. This standard Infographic setup emphasizes consistency, as required by principle (3). For a system, the Infographics are presented as a full screen panel. Each system has at least one Infographic, but might have unlimited compositions. An Infographic element can connect to another of the same type, representing a total screen change, with new graphs and actions (see the section IV for limitations in the scope of this work).

The second element are the graphs (bar graph, percentage bar graph, pie chart and line graph). Each graphic has its data range selection to plot and update intervals, allowing the user to generate as many visualizations for each data as desirable. Those basic types meet the principles (1) and (2) expectation that graphics that must easily compared and relevant for a use case. Also, it illustrates extensibility; any new variety of graph offered by the implemented language in the future can be added to the metamodel. Two elements provide data to the graphics - (i) the Data Source and (ii) the Formula. The Data Source is the element that points to the database storing the sensor's data. In this implementation, the Data Source is

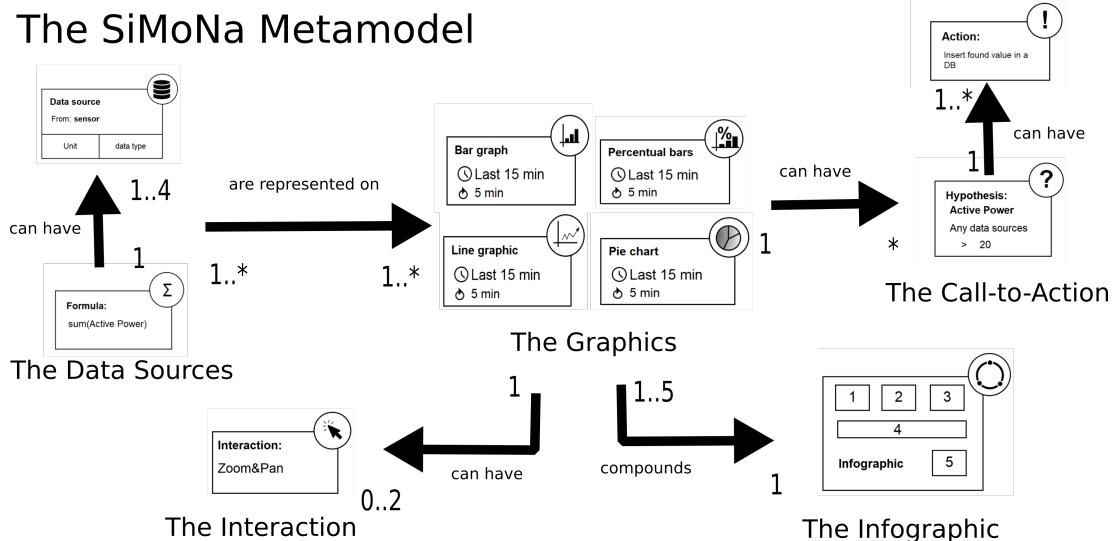


Fig. 1. The SiMoNa Metamodel. The arrows represent a interconnection possible between the elements. The number next to the arrow, the multiplicity of the element.

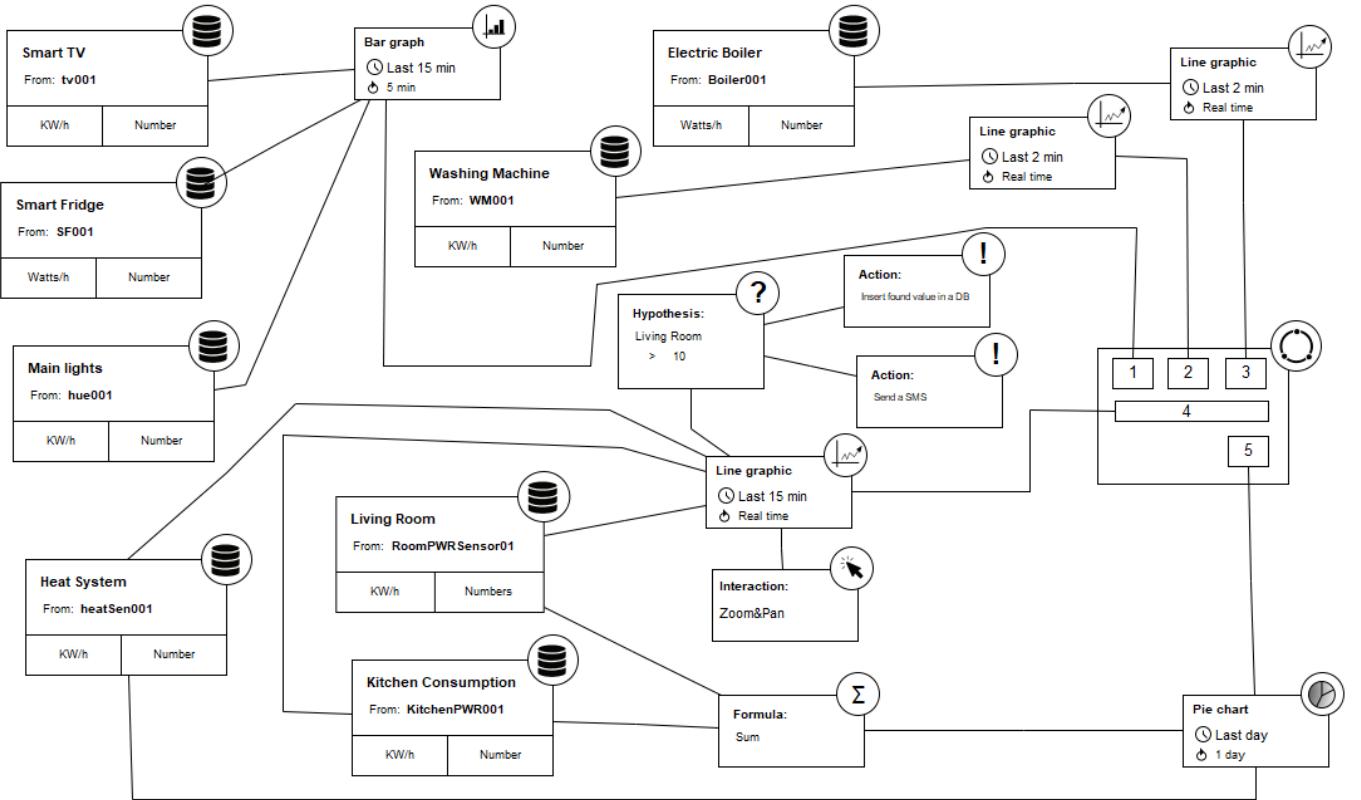


Fig. 2. A Smart Home energy efficiency scenario modeled in the SiMoNa DSML.

a JSON URL to fetch the data from an API/database. The Formula element is a processed Data Source. With Formula, the data plotted in a Graph can be parsed with a statistical method (average, median, or mode) or some complex formula made in AsciiMath [28]. In this proof-of-concept, Formula can have up to four elements, including other Formulae. The output of this Formula will be the information plotted in the Graph.

The next elements are related to hypothesis, thresholds, and actions. Those elements meet principles (4) and (5), as the user can explore and set triggers to the system. In the metamodel, the Hypothesis element is applied to a Graph to verify a specific threshold of a variable. With this element, the Action element performs a specific task when a condition is met in the Hypothesis. This action could be, for example, sending an email or an SMS, inserting the value (and its correlated variables) in a database, or a simple warning in the Infographic notification area.

The last element in the metamodel is the Interaction. This element adds an interaction ability to some graphs. There are various use cases, so the metamodel language supports both interactive (as in a tablet or computer) and non-interactive interfaces (as in static panel in a factory). As example, a home energy efficiency case (a device-focused energy efficiency monitoring scenario) was modelled in figure 2.

IV. CONCLUSION AND FUTURE WORKS

The complexity of IoT information systems require a fast and adaptable solution to handle data visualization. This paper proposes SiMoNa, a domain-specific modeling language (DSML) based on model-driven development (MDD) to provide visual information through Infographics to handle data that is generated from IoT systems. By using SiMoNa, it is possible to model and generate an Infographic system to visualize, compare and analyze data generated by an IoT system.

This is the first proof-of-concept implementation of the SiMoNa language. It is expected that some requirements will vary and/or improve during a real world application. Future iterations of the model may integrate new Infographic models that can easily be added to the metamodel as it has been designed to accommodate more than one base Infographic.

REFERENCES

- [1] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," *Mobile networks and applications*, vol. 19, no. 2, pp. 171–209, 2014.
- [2] Y. Sun, H. Song, A. J. Jara, and R. Bie, "Internet of things and big data analytics for smart and connected communities," *IEEE Access*, vol. 4, pp. 766–773, 2016.
- [3] F. J. Riggins and S. F. Wamba, "Research directions on the adoption, usage, and impact of the internet of things through the use of big data analytics," in *System Sciences (HICSS), 2015 48th Hawaii International Conference on*. IEEE, 2015, pp. 1531–1540.
- [4] J. K. Cleber Matos de Moraes, Djamel Sadok, "An iot sensors and scenarios survey for data researchers," *Journal of the Brazilian Computer Society*, in publishing.

- [5] J. W. M. C. P. B. J. Bughin, J. Manyika and R. Dobbs, "The internet of things: Mapping the value beyond the hyper," *McKinsey Global Institute*, June 2015.
- [6] V. Foteinos, D. Kelaïdonis, G. Poulios, P. Vlacheas, V. Stavroulaki, and P. Demestichas, "Cognitive management for the internet of things: A framework for enabling autonomous applications," *IEEE Vehicular Technology Magazine*, vol. 8, no. 4, pp. 90–99, 2013.
- [7] N. Li, M. Sun, Z. Bi, Z. Su, and C. Wang, "A new methodology to support group decision-making for iot-based emergency response systems," *Information Systems Frontiers*, vol. 16, no. 5, pp. 953–977, Nov 2014. [Online]. Available: <https://doi.org/10.1007/s10796-013-9407-z>
- [8] J.-D. Fekete, J. J. van Wijk, J. T. Stasko, and C. North, *The Value of Information Visualization*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–18.
- [9] C. G. García, B. C. P. G-Bustelo, J. P. Espada, and G. Cueva-Fernandez, "Midgar: Generation of heterogeneous objects interconnecting applications. a domain specific language proposal for internet of things scenarios," *Computer Networks*, vol. 64, pp. 143–158, 2014.
- [10] S. M. Pradhan, A. Dubey, A. Gokhale, and M. Lehofer, "Chariot: A domain specific language for extensible cyber-physical systems," in *Proceedings of the Workshop on Domain-Specific Modeling*, ser. DSM 2015. New York, NY, USA: ACM, 2015, pp. 9–16. [Online]. Available: <http://doi.acm.org/10.1145/2846696.2846708>
- [11] C. Ledur, D. Griebler, I. Manssour, and L. G. Fernandes, "Towards a domain-specific language for geospatial data visualization maps with big data sets," in *Computer Systems and Applications (AICCSA), 2015 IEEE/ACS 12th International Conference of*. IEEE, 2015, pp. 1–8.
- [12] M. Sevenich, S. Hong, O. van Rest, Z. Wu, J. Banerjee, and H. Chafi, "Using domain-specific languages for analytic graph databases," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1257–1268, 2016.
- [13] P. Klint, T. Van Der Storm, and J. Vinju, "Rascal: A domain specific language for source code analysis and manipulation," in *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*. IEEE, 2009, pp. 168–177.
- [14] E. Tufte and P. Graves-Morris, *The visual display of quantitative information.; 1983*. Cheshire, Connecticut, USA: Graphic Press, 2014.
- [15] P. Jacob and M. Jeannerod, *Ways of seeing: The scope and limits of visual cognition*. Oxford university Press, 2003.
- [16] C. Ware, *Information visualization: perception for design*. Elsevier, 2012.
- [17] A. Cairo, *Infografia 2.0*. Madrid, Spain: Alamut, 2008.
- [18] Electric Ireland. (2018) Wattless charges for business - explained. [Online]. Available: <https://www.electricireland.ie/business/help/efficiency/wattless-charges-for-business-explained>
- [19] C. E. Shannon and W. Weaver, "The mathematical theory of communication. 1949," *Urbana, IL: University of Illinois Press*, 1963.
- [20] L. Masud, F. Valsecchi, P. Ciuccarelli, D. Ricci, and G. Caviglia, "From data to knowledge-visualizations as transformation processes within the data-information-knowledge continuum," in *Information Visualisation (IV), 2010 14th International Conference*. IEEE, 2010, pp. 445–449.
- [21] N. Iliinsky, "On beauty," *Beautiful visualization: Looking at data through the eyes of experts*, pp. 1–13, 2010.
- [22] D. A. Dondis, *A primer of visual literacy*. Mit Press, 1974.
- [23] C. Atkinson and T. Kuhne, "Model-driven development: a metamodeling foundation," *IEEE software*, vol. 20, no. 5, pp. 36–41, 2003.
- [24] S. Kelly and J.-P. Tolvanen, *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008.
- [25] U. Frank, "Domain-specific modeling languages: requirements analysis and design guidelines," in *Domain Engineering*. Springer, 2013, pp. 133–157.
- [26] G. Team. (2018) Networking and telecommunications research group – gprt. [Online]. Available: <https://www.gprt.ufpe.br/gprt/>
- [27] J.-P. Tolvanen and S. Kelly, "Metaedit+: defining and using integrated domain-specific modeling languages," in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. ACM, 2009, pp. 819–820.
- [28] J. Gray, "Asciimathml: now everyone can type mathml," *MSOR CONNECTIONS*, vol. 7, no. 3, p. 26, 2007.

Visual Modeling of Cyber Deception

Cristiano De Faveri and Ana Moreira

NOVA LINCS, Department of Computer Science
 Faculty of Science and Technology, Universidade NOVA de Lisboa
 c.faveri@campus.fct.unl.pt, amm@fct.unl.pt

Abstract—Deception-based defense relies on deliberated actions to manipulate the attackers' perception of a system. It requires careful planning and application of multiple techniques to be effective. Therefore, deceptive strategies should be studied in isolation from the traditional security mechanisms. To support this goal, we develop DML, a visual language for deception modeling, offering three complementary views of deception: requirements model, deception tactics feature model, and deception strategy organizational. DML integrates goal-oriented requirements models and threat models to compose a comprehensive model considering the influences of developing deceptive mechanisms and the associated risks. The feasibility of DML is demonstrated via a tool prototype and a set of illustrative scenarios for a web system.

I. INTRODUCTION

Traditional approaches to cyber security have been continuously challenged by the ever-growing sophistication of attacks. Adversaries using massive software exploitation and social engineering techniques have proven to subvert boundary controllers, malware scanners, and intrusion prevention technologies, raising the need for new forms of defense that can influence attackers' decision [1]. One such means of influencing the attackers' move is *deception*. By presenting facts and fiction, defenders can provide misleading information and persuade adversaries to actively create and reinforce their perceptions and beliefs, engaging these in the defender's deception story. Deception is prevalent in attacks in cyberspace, but its scientific foundation as a critical component of an active cyber defense paradigm [2] has gained attention only very recently (e.g., [1], [3], [4]).

To be effective, a deceptive mechanism should continuously deceive attackers while minimizing both the interferences in the system operation and the risks of exposing real resources. To mitigate risks and avoid system interferences, many systems based on deception are designed to be completely isolated from the real systems (e.g., server honeypots [5]). However, these honey-systems have shown to be laborious to implement and maintain [6]. As for totally "fake systems", many tools currently exist to identify whether they are honey systems or not [6], [7]. In contrast, integrated deception approaches propose to add to the real systems interacting deception-based artifacts [8]. One incentive to incorporate deception techniques into the real system operation is to facilitate the production of more plausible and manageable deception mechanisms,

eliminating the need to fully reproduce the real system [8]. As the number of deceptive mechanisms spans through different points of a system, the greater the need to develop systematic approaches to specify these mechanisms.

The goal of this paper is to propose a systematic visual modeling approach to specify deception concerns during requirements elicitation and specification. The resulting contribution is two-fold: (i) assist engineers with a comprehensive language and toolset that address deception security concerns; (ii) separate traditional security requirements and threat models from deceptive solution mechanisms, allowing to compose distinct strategies to mitigate a set of threats, attack vectors and potential vulnerabilities in a system.

II. DML, A DECEPTION MODELING LANGUAGE

Deception Modeling Language (DML) is a visual language designed to model defense strategies based on deception. We envision that the key users of DML are software and security engineers who need to integrate deception with traditional security mechanisms in different layers of computations. DML can be used during the early stages of development (planning) or on established systems that require to incorporate deception into their operations. The primary objectives of DML are: (i) **identifying deception strategies early** in the development so that potential conflicts arising from system components and resources, and other deception strategies can be addressed; (ii) **integrating deception modeling with threat analysis** allowing to create deception strategies based on different elements of a threat model; (iii) **identifying distinct strategies based on deception** to be incorporated in the system (each strategy is composed of different mechanisms that mitigate one or more threats, attacks or vulnerabilities); (iv) **supporting risk analysis**, allowing designers to decide where to put their effort to enable a particular deception strategy in a system.

A. DML Metamodel

The abstract syntax of DML is represented by its metamodel and describes the relevant domain concepts. Figure 1 presents a concise version of the DML metamodel. The metamodel integrates three different model components: a deception requirements model (DREM), a deception tactic feature model (DTFM), and a deception strategy organizational model (DSOM). DREM represents the deception specification;

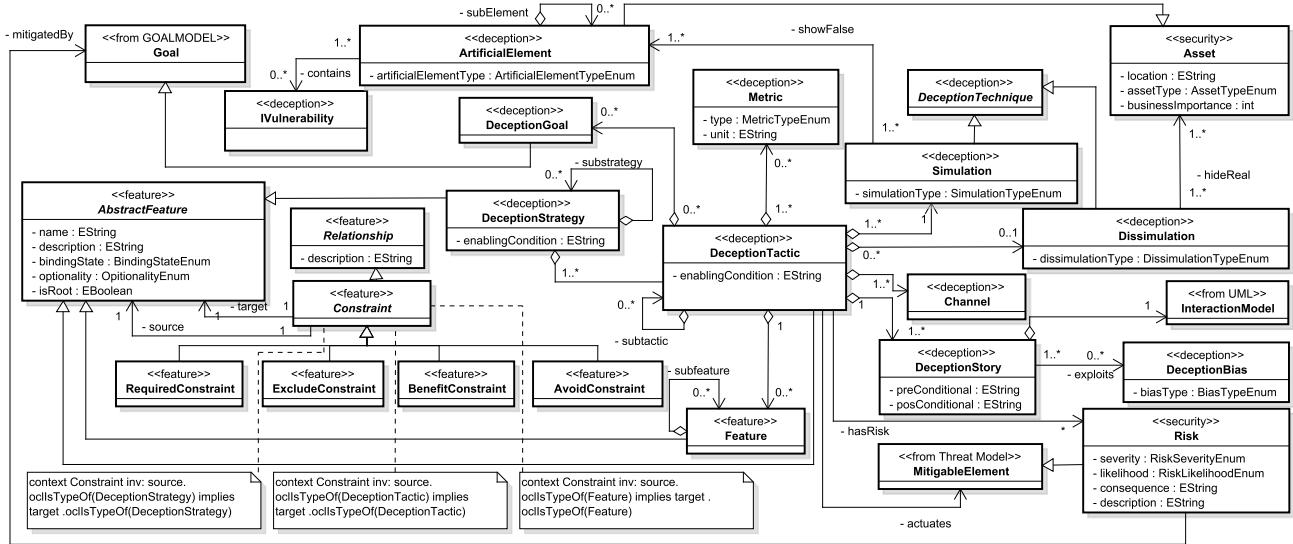


Fig. 1: DML metamodel

it uses concepts derived from general theory of deception and deception concepts applied to defense. A DTFM is a feature (Feature) model that captures common and variable properties (or features) of system families [9]. Finally, a DSOM organizes strategies (DeceptionStrategy) based on a set of tactics (DeceptionTactic) to be employed in a system.

A deception tactic is realized through functions, software components, scripts, configurations, context rules, and meta-protocols that prescribe the deception portray. A goal (DeceptionGoal) can be specified for a tactic. Our approach is well integrated with goal modeling methodologies (e.g., [10], [11]), which allows, for example, determining conflicts between a deception goal and a system goal. A deception tactic employs a technique (DeceptionTechnique), which specifies the Simulation and/or Dissimulation behavior of the system, based on the well-known deception taxonomy of Whaley [12]. A technique can simulate by creating an artificial element (ArtificialElement) and dissimulate by hiding a real asset. An artificial element may plant intentional controlled vulnerabilities (IVulnerability) in the system to entice attackers. IVulnerabilities and ArtificialElement are generalizations of real identified vulnerabilities and assets in the system. However, they may not have a conceptual correlation with the system assets or vulnerabilities. As artificial elements, they can be creatively invented to entice and manipulate the attackers' perception. We do not consider techniques that involve hiding the false and showing the real in our metamodel, since they are considered upholding techniques for simulation and dissimulation [4].

A tactic is dynamically described by one or more scenarios capturing its expected interactions with the adversaries. These scenarios are described by one or more stories (DeceptionStory) using models expressing dynamic behavior (InteractionModel), such as UML interaction diagrams. Stories can

be associated with the biases (DeceptionBias) that it exploits. This represents general information about which biases are supposed to be exploited by the defender.

A set of metrics (Metric) evaluates the deception tactics and strategies. Some metrics are quantitative and others are qualitative. Quantitative metrics are calculated by a formula, an indicator, an estimation method, or a measurement function. Qualitative metrics assign a qualitative value to a deception tactic, measuring a particular quality attribute. For example, the notions of plausibility and enticability¹ can be assigned a value from a scale ranging from "very low" to "very high". To calculate the metrics, a tactic must provide one or more channels of communication (Channel). A channel expresses the necessary requirements to collect information about the deception when engaged by an attacker. For example, which elements and data should be monitored and when.

In this context, risk analysis is the activity of analyzing the mechanisms and elements that constitute the strategy and evaluate their impact on the system operation. The importance of identifying and analyzing risks during deception modeling is twofold: (i) threats that are more risky can be prioritized during the design of a deception strategy; (ii) deception tactics can also pose risks that need to be acknowledged before their execution. The risk level is computed by combining the impact with the likelihood of an event occurring. A likelihood scale comprehends values like "very low", "low", "moderate", "high" and "very high", while the severity can be classified into "insignificant", "minor", "moderate", "major", "catastrophic", and "doomsday".

The central concept of DSOM and DTFM is the AbstractFeature from which DeceptionTactic, DeceptionStrategy, and Feature are derived. The attribute bindingState in Abstract-

¹A technical term used to define the enticement of a tactic.

Feature describes the phase in which the feature is bound, unbound or removed during its runtime life-cycle. An unbound feature means that it is part of the system and can be bound in the future. Removed features are applied on non-mandatory features. Constraints (Constraint) are refined into (i) require (RequireConstraint), representing a dependency relation from a node K to a node L , (ii) exclude (ExcludeConstraint), representing a mutually exclusive relation between a node K and a node L , (iii) benefit-from (BenefitConstraint), representing a contribution of the node K to a node L , and (iv) avoid (AvoidConstraint), meaning that when node K is enabled, L should not be enabled – failing to comply may lead to inconsistencies or an increase in the overall risk.

B. DML Notation

We design the DML concrete syntax by balancing expressiveness and simplicity while following the Moody's physics of notations (PON) principles [13]. Figure 2 presents the most relevant elements used by DML. The upper elements of the figure are used to construct a DREM. The lower elements are used to build a DSOM and a DTFM. We explicitly discriminate elements specifically related to deception, using a special notation (a dark circle with a "D" inside). This facilitates the integration with other models (e.g., goal oriented models and threat models), allowing tools to create separate views for deception elements. DSOM and DTFM model elements use the same elements of traditional feature models (rectangle), but a stereotype is included to discriminate the semantics of the element against descendants of AbstractFeature, namely «strategy», «tactic», and «feature». Similarly, we use a discriminator for each association, represented by «requires», «excludes», «helps»(benefits-from), and «avoids», according to the language specification. To show the feasibility of our approach, we develop the Deception Modeling Tool (DMT) that can be accessed at <http://tiny.cc/wm2kpy>.

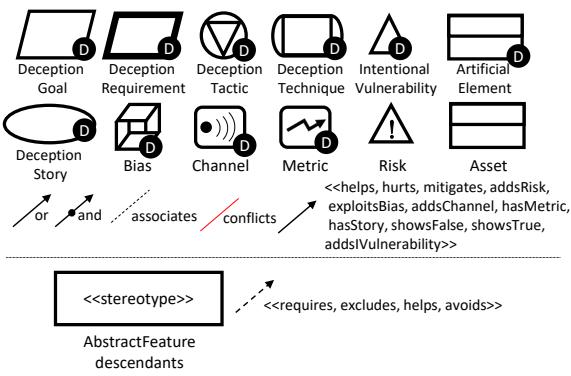


Fig. 2: DML partial notation

III. DML APPLICATION

Consider the scenario where the board of a company handles sensitive data using a web software system that allows: (i) saving flat log files containing data on the system operation (e.g.,

critical transactions, warnings, errors, etc.); (ii) user authentication using a login and password stored in a remote database using cryptography methods; (iii) cookies to control sessions and user preferences; (iv) system configuration settings being performed by administrators using an administration page, accessible from the local network.

The goal is to model a deception strategy containing several tactics to mitigate the threats and potential vulnerabilities that can jeopardize the system operation. Let's consider that a threat model defining threats, attack vectors, potential vulnerabilities and critical assets already exists. Fig. 3 shows five threats (T1 to T5) with the following scenarios: (T1) System malfunction by parameter manipulation: values of URL parameters changed to cause system failure, (T2) Password leaked out: database accessed by attackers, which accounts, including passwords, are exfiltrated, (T3) System failure by cookie manipulation: manipulation of cookies (cookie poisoning) by changing their content to unexpected values in an attempt to cause system failure, (T4) System malfunction by page field manipulation: hidden field pages manipulated to cause system failure, and (T5) System administration function authentication bypassed: administration page being accessed by unauthorized users, including internal malicious users.

Each threat is associated to one or more deception tactics to mitigate the threat or capture any misbehavior that will lead to triggering an alert in the system. Honeyparameters (tactic to mitigate threat T1) add fake parameters to a URL. When the content of these parameters is different from the expected values, we can add deceptive actions, such as slowing down the response, or simulating a failure to provoke the attacker to continue the endeavor. To threat T2, we associate two possible tactics: the use of honeywords or ersatzpasswords [14]. These deceptive tactics are used to mitigate off-line password cracking. Honeywords add fake passwords along with one correct password to cause confusion on adversaries in determining which one is the correct password in case of repository exfiltration. Ersatzpasswords use a machine-dependent function and a deceptive mechanism to cover real passwords and enhance the security in case of password exfiltration. Honeycookies (T3) and Honeyfields (T4) are similar to honeyparameters, but it adds false cookies and false hidden fields on web pages, respectively. Finally, we employ the tactic Fake Account Authentication for threat T5. This tactic is subdivided into LogInstrumentation and Honeyaccount. Honeyaccount creates one or more bogus accounts in the database tables that store user administrators. LogInstrumentation provides proper instrumentation on log files to add fake data containing the database URL and a user name (account) that identifies a user administrator. Any attempt to use this account represents a violation of security that requires further investigation. Of course, legitimate users require a method to separate bogus records from real ones. This can be accomplished, for example, by keeping indexes of bogus records in a safer place other than the log file.

We focus on the construction of a DREM for the tactic Fake Account Authentication, as illustrated in Fig. 3. We

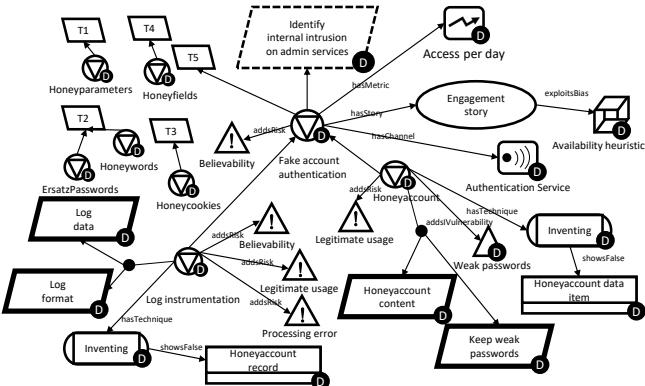


Fig. 3: Partial DREM: Fake Account Authentication specification

will use this example to illustrate how the DML is used to model a tactic, however, for the sake of space, we omit the construction of the DSOM and the DTFM. In general, a tactic is executed to achieve a goal. This is expressed by the softgoal "Identify internal intrusion on admin services". The tactic Log instrumentation uses the Inventing technique that will show a Honeyaccount record on log file. This tactic has two requirements: Log data and Log format. Log data describes which data should be added to the log file and when this data should be added. Log format describes the message format that will be recorded into the log file. These requirements can be expressed informally using natural language or formally using a language such as Linear Temporal Logic (LTL) [15]. For example, log entries should use the same format of regular messages since a different pattern can be suspicious. Log instrumentation poses some risks, namely Believability, Legitimate usage, and Processing error. Believability indicates the probability of an attacker not believing in the information that he is observing in the log file. For instance, an enticing name for a user, such as admin-test could be more plausible, leading to low-risk likelihood. However, the impact can be high, since the attacker will remain undetectable if he suspects the data is fake. Similarly, the risk Legitimate usage indicates the risk of legitimate users reaching the bogus data and using it. By using some mechanism to discriminate real from fake entries, this risk is low with no relevant impact. Similarly, the risk Processing error considers the log being read by tools considering the artificial honeyaccount information. This risk is also mitigated using the mechanism for discrimination. Notice that new requirements can be described to mitigate the risks identified during the process of constructing a DREM.

The Honeyaccount tactic also has associated a simulation technique that creates a new honeyaccount item in the database (Inventing showsFalse Honeyaccount data item). Two requirements are also associated to this tactic: Honeyaccount content and Keep weak passwords. Honeyaccounts contents describe how honeyaccounts should be created in the system. Keep weak passwords intends to describe how to associate weak passwords to this honeyaccount just to keep a plausible sce-

nario for an attacker. This leads to an intentional vulnerability in the system, expressed by the element Weak passwords. Honeyaccount tactic is associated with a risk of legitimate users using this account for authentication (Legitimate usage). By choosing user accounts that cannot be created via normal procedures in the system, the probability of a legitimate user being authenticated by a honeyaccount is considered low. Of course, legitimate users authenticating with honeyaccounts is considered a distrustful behavior. Authentication Service is the channel where the use of a honeyaccount will be verified. The Engagement story is straightforwardly described using an interaction diagram, as illustrated in Fig. 4. The attacker engages in the deception by authenticating in the administrator service page using a honeyaccount. The response is a deceptive message indicating that the service is down for maintenance procedures. The diagram describes a setup phase, when the honeyaccount is stored in the database and the log is instrumented with deceptive information. Finally, the tactic is associated with the metric Access per day. This metric is a quantitative metric that computes the number of authentication attempts using a honeyaccount.

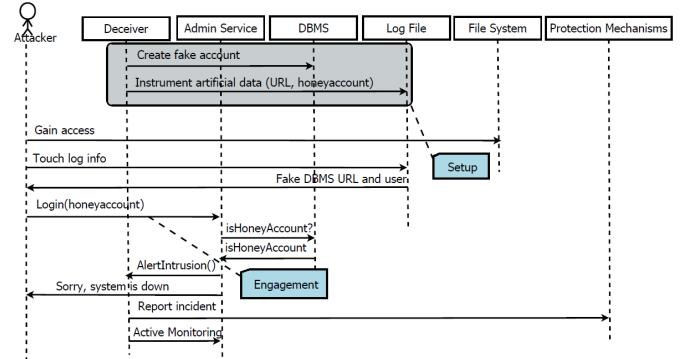


Fig. 4: Administrator Honeyaccount engagement story description

IV. CONCLUSIONS AND FUTURE WORK

DML is a visual language to model deception strategies and tactics in a software system. It assists engineers to address deception security concerns since early stages of software development by separating traditional security requirements and threat models from deceptive solution mechanisms. Although the concepts described in the DML metamodel are grounded on well-accepted deception theory, finding the proper balance between abstraction and resulting usefulness is always challenging. While a higher level of abstraction tends to be more expressive, it can also be too general and not meaningful enough for the modeler. To mitigate this problem, we are currently planning further evaluation of the DML and DMT with other technologies (such as IoT) and conducting an empirical evaluation with real users.

Acknowledgments. The authors are grateful to CAPES (process 0553-14-0), FCT and NOVA LINCS Research Laboratory (Ref. UID/CEC/04516/2013).

REFERENCES

- [1] K. E. Heckman, F. J. Stech, R. K. Thomas, B. Schmoker, and A. W. Tsow, *Cyber Denial, Deception and Counter Deception*. Springer, 2015.
- [2] D. E. Denning, “Framework and principles for active cyber defense,” *Computers and Security*, vol. 40, no. December, pp. 108–113, 2014.
- [3] N. C. Rowe and J. Rushi, *Introduction to Cyberdeception*. Springer, 2016.
- [4] S. Jajodia, V. S. Subrahmanian, V. Swarup, and C. Wang, *Cyber deception: Building the scientific foundation*. Springer Nature, 2016.
- [5] L. Spitzner, *Honeypots: Tracking Hackers*, vol. 1. Addison-Wesley Reading, 2002.
- [6] X. Chen, J. Andersen, Z. Morley Mao, M. Bailey, J. Nazario, Z. Mao, and M. Bailey, “Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware,” in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 177–186, 2008.
- [7] T. Holz and F. Raynal, “Detecting honeypots and other suspicious environments,” in *Proceedings from the 6th Annual IEEE System, Man and Cybernetics Information Assurance Workshop, SMC 2005*, vol. 2005, pp. 29–36, IEEE, 2005.
- [8] M. H. Almeshekah, *Using deception to enhance security: A Taxonomy, Model, and Novel Uses*. PhD thesis, Purdue University, 2015.
- [9] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, “Feature-oriented domain analysis (FODA) feasibility study,” tech. rep., DTIC Document, 1990.
- [10] A. Dardenne, A. Van Lamsweerde, and S. Fickas, “Goal-directed requirements acquisition,” *Science of computer programming*, vol. 20, no. 1-2, pp. 3–50, 1993.
- [11] E. S. K. Yu, “Towards modelling and reasoning support for early-phase requirements engineering,” in *Requirements Engineering, 1997, Proceedings of the Third IEEE International Symposium on*, pp. 226–235, IEEE, 1997.
- [12] B. Whaley, “Toward a general theory of deception,” *The Journal of Strategic Studies*, vol. 5, no. 1, pp. 178–192, 1982.
- [13] D. L. Moody, “The ‘physics’ of notations: toward a scientific basis for constructing visual notations in software engineering,” *Software Engineering, IEEE Transactions on*, vol. 35, no. 6, pp. 756–779, 2009.
- [14] M. H. Almeshekah, C. N. Gutierrez, M. J. Atallah, and E. H. Spafford, “ErsatzPasswords: Ending Password Cracking and Detecting Password Leakage,” in *Proceedings of the 31st Annual Computer Security Applications Conference*, pp. 311–320, ACM, 2015.
- [15] A. Van Lamsweerde, “Requirements engineering: from system goals to UML models to software specifications,” 2009.

Milo: A visual programming environment for Data Science Education

Arjun Rao*, Ayush Bihani†, Mydhili Nair‡

Department of Information Science and Engineering

Ramaiah Institute of Technology

Bangalore, India

Email: *mailto:arjunrao@gmail.com, †bihani37@gmail.com, ‡mydhili.nair@msrit.edu,

Abstract—Most courses on Data Science offered at universities or online require students to have familiarity with at least one programming language. In this paper, we present, “Milo”, a web-based visual programming environment for Data Science Education, designed as a pedagogical tool that can be used by students without prior-programming experience. To that end, Milo uses graphical blocks as abstractions of language specific implementations of Data Science and Machine Learning(ML) concepts along with creation of interactive visualizations. Using block definitions created by a user, Milo generates equivalent source code in JavaScript to run entirely in the browser. Based on a preliminary user study with a focus group of undergraduate computer science students, Milo succeeds as an effective tool for novice learners in the field of Data Science.

I. INTRODUCTION

Over the last four years, we have seen a lot of growth in the use of Data Science in modern applications. According to LinkedIn’s 2017 report [1], the top ranked emerging jobs in the U.S. are for Machine Learning Engineers, Data Scientists, and Big Data Engineers. The report also highlights that while the number of roles in the Data Science domain has risen many fold since 2012, the supply of candidates for these positions is not meeting the demand.

The most common path taken towards understanding Data Science is still through university programs, online courses and workplace training. We surveyed popular online courses in the domain using Class Central [2] and found that most courses either require prior programming experience in Python or use tools like MATLAB, Octave, R, Weka, Apache Spark, etc. which can be intimidating to non-computer science majors.

In the general field of computer science education, there have been many efforts for introducing fundamental concepts of programming to beginners through visual tools. Examples include those on Code.org or MIT’s Scratch project [3]. However there have been fewer efforts in building tools for introducing concepts in Data Science and Machine Learning to non-programmers.

In this paper, we present “Milo”, a web based visual programming environment for Data Science Education. Our primary aim when designing Milo, was to build a platform that is approachable to non-computer science majors, and allows students to self-learn concepts of Data Science and Machine Learning. To support these goals, we built Milo to work in the

browser, and use a block-based programming paradigm that is suitable for novices and non-programmers. The main interface of Milo is shown in Figure 1, and consists of graphical blocks which abstract implementations of various concepts covered in typical Data Science courses. Supported concepts include basic statistics, linear algebra, probability distributions, ML algorithms, and more. The workspace is built using Blockly¹ and the blocks have a similar look and feel to that of Scratch [3].

Our target audience for Milo, is two fold. On one hand we target students from high-school to undergraduate students in non-computer science fields. For students who are not familiar with programming but have an understanding of basic concepts in linear algebra, and statistics, we feel that Milo is a good avenue for getting hands on exposure to using these concepts in solving practical problems, and getting exposure to the world of programming in an intuitive and visually rich manner. On the other hand we target faculty and educators, who design introductory courses for non-programmers in the fields of Data Science, Machine Learning and Linear Algebra.

The rest of this paper is organized as follows. Section II highlights related work in this domain, particularly visual programming environments and tools that we referred to. In Section III, we talk about Milo’s programming model and compare this with other popular approaches. This is followed by details of our implementation in Section IV. Section V summarizes a preliminary evaluation of Milo via a user study. We then note a few limitations of our work in Section VI, and present a road map for the future (Section VII) and our conclusions (Section VIII).

II. RELATED WORK

Visual programming environments are alternatives to text based programming, having logic constructs expressed using graphical operators and elements. This is not a new concept, as prior work on such forms of programming date back to the 90s. Work done by [4], [5], and [6], have influenced many projects from the 90s to present times, showing that visual programming environments are commonly employed in practice.

¹<https://developers.google.com/blockly/>

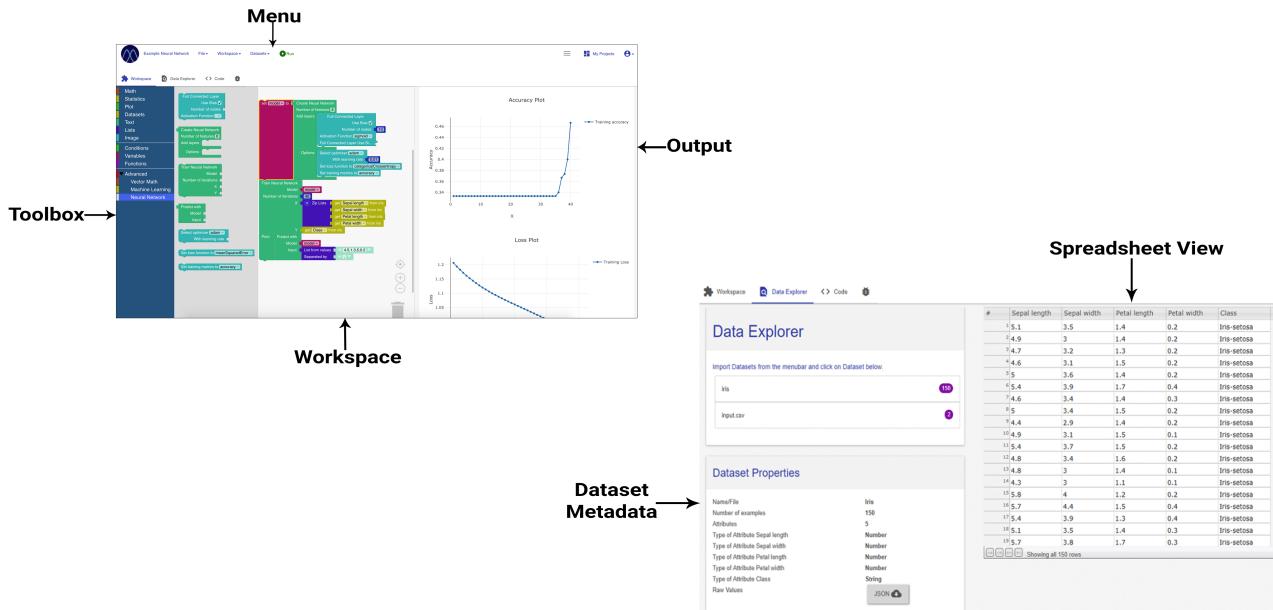


Fig. 1. The top left screenshot shows the Milo IDE Interface consists of the top menu, a toolbox that holds blocks organized by category, the workspace for building block based programs, and the output pane. The bottom right screenshot shows the data explorer with the Iris dataset loaded in a spreadsheet like view.

Scratch [3] is a visual programming language and online community targeted primarily at novice users, and is used as an introductory language to delve into the field of programming through blocks. Our main motivation for choosing a block-based design for Milo was Scratch, due to it's proven track record for being a popular introductory tool for non-programmers to get involved with computer science. The user interface of Scratch follows a single-window, multi-pane design to ensure that key components are always visible. The success of this approach motivated us to build a single window IDE in Milo, that allows execution of programs along side the workspace used to create them. This prevents distraction for users and presents all the important aspects of the IDE in one place.

According to Zhang et al. [6], a single environment for researchers to manage data and perform analysis, without having to learn about multiple software systems and their integration, is highly effective. Thus Milo borrows these ideas and includes a Data Explorer for viewing and understanding datasets in a spreadsheet like format, along with the main block workspace that is used to perform operations on data or train ML models. (See Fig 1)

BlockPy [7] is a web based python environment built using Blockly with a focus on introductory programming and data science. This is primarily done by integrating a host of real-world datasets and block based operations to create simple plots of data. However we found that BlockPy is primarily suited to give a gentle introduction to Python and falls short of the requirements of a full-fledged Data Science course.

Another popular tool used for teaching Data Science is

Jupyter Notebooks². While it is a great tool for exploratory data analysis and quick prototyping, we found that Jupyter notebooks are more suited for Computer Science majors, and those who are familiar with concepts of programming in general, and more specifically those who know Python or Julia.

III. COMPARISONS BETWEEN PROGRAMMING MODELS

Milo uses a block based programming model. This approach to programming is unlike that of popular visual tools for Machine Learning like Rapid Miner³ or Orange⁴, as they follow a dataflow approach to programming. In this section, we focus on the programming model of Milo, and compare this with that of tools that use a dataflow approach. Additionally, we compare Milo with Scratch, and note their differences.

When compared with Scratch, Milo's programming model may seem very similar in terms of look and feel. This is because they are both rendered using Blockly, however, the styles, blocks, and their connections are designed for different use cases, and hence the language vocabulary and block patterns are different. Unlike Scratch, Milo does not use an event driven model. This is because, we do not have sprites, or a stage with graphical objects that interact with one another. Instead, Milo uses a sequential approach to programming, where blocks are executed from top to bottom in the workspace, ie. blocks placed above others will be executed first. Additionally, Milo generates syntactically correct source code from the block definitions, and this is presented in

²<https://jupyter.org>

³<https://rapidminer.com/>

⁴<https://orange.biolab.si/>

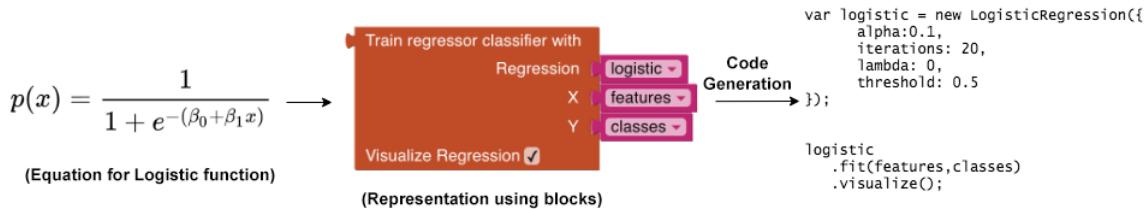


Fig. 2. Shows how a machine learning concept such as a classifier using logistic regression is represented using blocks on Milo, along with the code it generates.

the code tab of the UI. During our prototyping stages, we found that using such a model makes the transition to real programming languages after Milo, fairly intuitive. The block constructs and their respective translations in JavaScript or Python are easily comparable and the sequential flow of execution is preserved after the translation.

When it comes to dataflow paradigms, we feel that while such paradigms are intuitive in understanding the transformations from input to output, it is less useful in understanding the internal steps of this transformation. It makes Machine Learning algorithms seem like black boxes to novice students, obscuring implementation details. In Milo, students can drag a single block, such as the one shown in Fig. 2, that trains a logistic regression based classifier using the given input, and produces an animated visualization of the training steps, which is similar to the black box like approach of dataflow programming. However they can also go a step further and build this block themselves by using primitive blocks for manipulating input vectors, and math operators like exponential functions or logarithms. Thus novice students would first learn concepts using built-in high-level blocks, and then figure out how to build these algorithms themselves using primitive blocks that they assemble from scratch.

IV. IMPLEMENTATION

We implemented Milo's block language using Google's Blockly library, which is used to build visual programming editors. The main user interface of Milo, as shown in Figure 1, consists of the workspace where block based programs are assembled, the output pane, a menu bar that lets users switch between the workspace, the data explorer, which is a space for viewing datasets in a spreadsheet like format, and a tab that shows generated code. The tool also includes a few popular datasets that are used in introductory ML courses.

In Milo, all programming constructs and implementations of various Data Science concepts are represented as interconnecting drag and drop blocks. They are the basic primitives for building any program on the platform. Connections between blocks are constrained such that incompatible blocks cannot be connected together. This allows us to generate syntactically correct code from block representations and prevent logical errors. Figure 2 is an example of how a machine learning concept like Logistic regression is represented through blocks and translated to source code.

TABLE I
TYPES OF BLOCKS IN MILO

 Create Neural Network Number of features 0 Add layers Options	Chained Input Blocks have space for chaining a number of supplementary input blocks (such as the Add Layers input in the block on the left), and are used to create dynamically defined objects. Examples include creating different neural network architectures by chaining neural network layer definitions one below the other or for creating multiple plots by chaining plot definitions.
	Compute blocks are those that have a notch on the left that represents a return value connection. These Blocks optionally take inputs and always return a value that is the result of some computation. The blocks may have additional options for advanced usage that is indicated by presence of a gear icon, on the block.
	Operation blocks are those that represent a single operation/function that does not return any value. These blocks can take input and additionally may transform their input but they do not return any value. The notches above and below the block are used to chain operations to execute in a particular order or to act as inputs to Chained Input Blocks

The blocks result in generation of syntactically correct Javascript code, which is used for execution on the web. We used tensorflow.js⁵ for implementing low-level math operations like matrix multiplications, vector manipulation, etc. Table I illustrates the various types of blocks available in Milo.

The Milo Platform consists of a NodeJS⁶ based web server, which acts as the backend, and the frontend for the platform is written in AngularJS⁷. The projects created using Milo, are stored as XML documents in a MongoDB database⁸. As part of the client side code, we include an execution library, which exposes high-level APIs for implementations of various Machine Learning algorithms, similar to what scikit-learn⁹ does in Python. The pre-made blocks for algorithms

⁵<https://js.tensorflow.org>

⁶<https://nodejs.org>

⁷<https://angularjs.org>

⁸<https://www.mongodb.com>

⁹<http://scikit-learn.org>

like KMeans, KNN, etc are translated to calls to functions in this execution library, and these in-turn are implemented using tensorflow.js and custom javascript code. These pre-made ML algorithm blocks also come with corresponding blocks to generate visualizations that show models being updated with each iteration of training, such as animated decision boundaries, or clustering data points in real time with iterations of training. Additionally, for some algorithms, like KNN, there are interactive visualization blocks that allow users to place a new test point on the 2d plane showing a scatter plot of the training dataset and see in real time what class the model might assign to this point, and what neighbours were considered. These interactive visualizations and other plotting functions in Milo are implemented using D3.js¹⁰.

V. PRELIMINARY EVALUATION

In order to evaluate our implementation, we conducted a user study, with a focus group of 20 undergraduate computer science students.

A. Study Setup

Participants were selected using a convenience sampling approach from a class of students who were taking their first introductory course in Machine Learning. The class follows the book Introduction to Machine Learning by Ethem Alpaydin [8]. Prior to the study we asked participants, to report their familiarity with various ML concepts, and their programming experience. We found that only 10% of participants reported that they would consider themselves more-comfortable with programming, while 55% considered themselves less-comfortable, with 25% of participants reporting that they had never programmed in Python/R or Julia before. In order to evaluate the usefulness of Milo in a course, we asked participants of the study to take a model class on Machine Learning that uses Milo as part of the pedagogy via a flipped classroom approach [9]. During the class participants used Milo, to perform clustering using K-Means on the Iris dataset [10].

After the class we administered a post-study questionnaire. The questionnaire asked students to rate various features of Milo that they tried, in terms of usefulness and ease of use, along with their perceived level of understanding K-Means clustering after the flipped-classroom activity. They were also asked open-ended questions that prompted feedback about various activities done as part of the class.

B. Study Results

- As the participants were taking a course on machine learning which followed a traditional classroom model, their experience with a flipped classroom model using Milo lead them to have highly positive sentiments.
- 90% of participants reported that visualizations were very easy to create using Milo and supplemented their understanding of the concepts.

¹⁰<https://d3js.org/>

- The study was mainly preliminary in nature, to evaluate the tool, in terms of usefulness and whether or not it met the requirements of students learning Machine Learning concepts for the first time, and based on the survey responses, we found that 70% of students felt the tool would be very useful for novice learners.

VI. LIMITATIONS

Due to the preliminary nature of our user study, our focus group size was limited. The students in the study had some level of prior-programming experience as they had taken at least one formal programming course. Considering that this was an initial study, we chose computer science students as our participants, because we felt they would be in a position to evaluate the merits of the tool in terms of what works, and what is missing. However a real test for the tool will only be when we conduct a user study with non-computer science students. As the main focus is education, Milo is not intended to be used for training and developing production ML models. The platform does not support advanced neural networks such as LSTM, GRU, CNN etc. Large datasets that exceed a few million rows may slow down browsers and may not be suitable for use in Milo.

VII. FUTURE WORK

Our goal with Milo is to help learners understand complex concepts using a simple visual approach. Concepts such as neural networks, and multivariate distributions need to be explained in an intuitive way to new learners. Keeping this in mind, the next iteration of Milo will include interactive visualization for neural networks, support for multinomial distributions, multivariate gaussians, etc. to make these concepts more approachable to beginners. To improve code re-usability, the platform will let users download generated code and results which can then be embedded in external blogs or other websites. While our current security model prevents, to a large extent, execution of code that may be harmful or malicious in nature, we are working on enhancing security by including a more robust execution sandbox for code that runs in the browser.

VIII. CONCLUSION

In this paper, we present Milo, a novel visual language targeting new learners in the field of Data Science and Machine Learning. We present our implementation of Milo, that uses modern web technologies to create a completely browser based platform for Data Science Education. Through our preliminary user study, we show that a visual programming environment is an effective platform for introductory courses on Data Science. Additionally, we establish a direction for future work in improving Milo.

CODE FOR PROTOTYPE

To facilitate research and further evaluation of our work, we have released our code on GitHub under an open source license, and can be found at <https://miloide.github.io/>.

REFERENCES

- [1] L. E. G. Team, "Linkedin's 2017 u.s. emerging jobs report," December 2017.
- [2] Class central: A popular online course aggregator. [Online]. Available: <https://www.class-central.com/>
- [3] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The scratch programming language and environment." *ACM Transactions on Computing Education*, vol. 10, no. 4, pp. 1–15, Nov 2011.
- [4] A. A. Disessa and H. Abelson, "Boxer: A reconstructible computational medium." *Communications of the ACM*, vol. 29, no. 9, pp. 859–868, Sept 1986.
- [5] G. EP., "Visual programming environments: Paradigms and systems," 1990.
- [6] Y. Zhang, M. Ward, N. Hachem, and M. Gennert, "A visual programming environment for supporting scientific data analysis." *Proceedings 1993 IEEE Symposium on Visual Languages*., 1993.
- [7] A. C. Bart, J. Tibau, E. Tilevich, C. A. Shaffer, and D. Kafura, "Blockpy: An open access data-science environment for introductory programmers," *Computer*, vol. 50, no. 5, pp. 18–26, May 2017. [Online]. Available: <doi.ieeecomputersociety.org/10.1109/MC.2017.132>
- [8] E. Alpaydin, *Introduction to machine learning*. The MIT Press, 2010.
- [9] M. B. Gilboy, S. Heinerichs, and G. Pazzaglia, "Enhancing student engagement using the flipped classroom," *Journal of nutrition education and behavior*, vol. 47, no. 1, pp. 109–114, 2015.
- [10] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of human genetics*, vol. 7, no. 2, pp. 179–188, 1936.

A Usability Analysis of Blocks-based Programming Editors using Cognitive Dimensions

Robert Holwerda

*Academy of Information Technology and Communication
HAN University of Applied Sciences
Arnhem, The Netherlands
robert.holwerda@han.nl*

Felienne Hermans

*Dept. of Software and Computer Technology
Delft University of Technology
Delft, The Netherlands
f.f.j.hermans@tudelft.nl*

Abstract—Blocks-based programming holds potential for end-user developers. Like all visual programming languages, blocks-based programming languages embody both a language design and a user interface design for the editing environment. For blocks-based languages, these designs are focused on learnability and low error rates, which makes them effective for education. For end-user developers who program as part of their professions, other characteristics of usability, like efficiency of use, will also be important. This paper presents a usability analysis, supported by a user study, of the *editor design* of current blocks-based programming systems, based on the Cognitive Dimensions of Notations framework, and we present design manoeuvres aimed at improving programming time and effort, program comprehension and programmer comfort.

Keywords—blocks-based languages, end-user development, programmer experience, cognitive dimensions

I. INTRODUCTION

With the success of blocks-based programming languages as an educational tool to teach novices about programming, the question arises if blocks-based programming can become an alternative to text-based languages in fields outside of education [1][2][3]. One promising area for blocks-based tools is the field of end-user programming. Because most end-user programmers will have little or no formal training in programming [4], the learnability of blocks-based programming is likely to benefit these end-user programmers as much as it benefits novices in education. Other than that, different kinds of end-user programmers will have different goals and requirements (see e.g. Table 1 in [5]), and this paper focuses on the usability requirements of *professional* end-user programmers.

In [2], we describe three reasons why the needs of professional end-user programmers are likely to differ from the needs of students during an introductory programming course: (1) for end-user programmers who program as part of their jobs, *long-term usability* can be very important in addition to initial learnability, as they reuse their programming skills in multiple projects, over a long period of time; (2) projects may grow *larger in size and complexity*; and (3) they need *access to more functionality*, including the ability to use multiple languages. Each of these three reasons can impact the user interface design of blocks-based programming tools for professionals. Improving *long-term usability* might require design changes geared toward efficiency of use, error reporting and fixing, and programmer comfort. Improving support for *large projects* might require new features to deal with search, navigation and (re)structuring the program. Allowing *access to more functionality* could force a rethinking of user interface features that depend on the language (and the set of run-time capabilities) being small or simple.

Any effort to adapt the design of blocks-based editors to professional end-user programming would benefit from a deep understanding of the current strengths and weaknesses of the blocks-based programming experience. This paper, therefore, presents an analysis of the usability of blocks-based programming editors. Results from a user study augment the analysis. The goal of this paper is to reveal design opportunities and priorities for bringing blocks-based language editing to end-user programming professionals, such as interaction designers, data journalists or system operators.

The conceptual framework for the analysis is the Cognitive Dimensions of Notation (CDN) [6], chosen for three reasons. First and foremost, the CDN dimensions are geared towards the evaluation of interactive programming tools. The framework has been used to evaluate multiple programming tools [7][8], including a blocks-based language, App Inventor [9]. Second, the CDN framework is less (than e.g. [10]) about quality judgments, and more about nuanced design trade-offs. And third, it explicitly supports distinguishing between different *layers* of an interactive notational system that can be analyzed separately.

The notion of layers in the CDN framework can be used to distinguish between the editor layer and the language layer [11]. This distinction is helpful for considering the possibility of a single blocks-based editor design for multiple different languages. Some classes of professional end-user programmers would be best supported by multiple (domain-specific) languages. A data journalist might need languages for database querying, statistical processing and/or visualization. When prototyping, a user interface designer might need declarative languages for UI specification and styling, and imperative languages for event handling and client-server communication. For users of multiple languages, having to learn a new blocks editor for each new language would decrease the usability of their complete toolset. For this audience therefore, we focus the analysis in this paper on the editor layer, in order to obtain results that are relevant to the design of blocks-based editors for multiple languages.

II. RESEARCH QUESTIONS

To investigate which design improvements will be relevant to professional end-user programmers, we answer three research questions for five cognitive dimensions. The five dimensions, selected by two criteria described below, are: *diffuseness*, *role-expressiveness*, *viscosity*, *secondary notation*, and *visibility*. The three research questions all center on *generic aspects of the user interface* of blocks-based programming editors. These are aspects that users would expect to stay consistent in a blocks-based editor that supports multiple languages. Not all user-interface design for a blocks-based

language is in the editor layer: For example, while *having* labels on input slots is an aspect of the editor design, the *wording* of the labels of a particular block is an aspect of the design of the language that defines the block. This means that the wording of labels will not feature in this analysis, but the existence of such labels will. We will term such generic aspects of the user interface of blocks-based programming editors *editor properties*. Examples of editor properties include: the 2D placement of blocks, the palette, the shapes of blocks, the visual design, the interaction mechanisms like drag-and-drop and drop-down menus, the affordances for those interactions, and features like duplicating, deleting, selecting, adding comments. Fig. 1 introduces terms that this paper uses to describe components of blocks-based editors.

These are the three research questions, with RQ1 and RQ2 providing input to RQ3:

RQ1: What editor properties affect the dimension? Are these properties problematic or beneficial to the user experience in this dimension? Are there properties that could become problematic if the editor were to be used for a long time, with a larger language, or with large programs? The CDN framework is designed to be a descriptive framework, providing a vocabulary for analysis and discussion [6]. This research question embodies this aspect of the framework.

RQ2: Which results from the user study are relevant to this dimension? The CDN framework was augmented, in [12], with a standard questionnaire, aimed at end-users, allowing empirical studies based on the framework.

RQ3: What design manoeuvres could improve the user interface of blocks-based editors with regard to this dimension? The CDN framework describes, in [11], several examples of design manoeuvres that can improve a system for one dimension, but also entail likely trade-offs in other dimensions.

For this analysis, we select the dimensions according to two criteria: First, we select the dimensions that are most relevant to the editor layer. Some dimensions, like *closeness of mapping* and *abstraction* are much more applicable to the language layer than to the editor layer, and are, therefore, not included in the current analysis.

Second, since the design of blocks-based editors is already aimed at learnability and low error rates [13][14], we select dimensions that are most relevant to other usability characteristics. Usability has five generally accepted characteristics, and besides learnability and error rates, these are efficiency of use, memorability, and satisfaction [15]. In [16], the author proposes a similar set of characteristics, that is specific for programming languages: learnability, error rates, programming time/effort, program comprehension, and programmer comfort. He also lists which cognitive dimensions affect which of those characteristics, allowing us to select dimensions that are relevant to programming time/effort, program comprehension, and programmer comfort.

Within the broad range of blocks-based languages, we focus our analysis on the two families that dominate the current educational use of blocks-based languages: (1) the Scratch family, consisting mainly of Scratch [13], Snap! [17], and the upcoming GP [18], and (2) the Blockly family: the languages created with the Blockly [14] toolkit, with MIT’s App Inventor [19] as a very prominent example. These two families share many user interface design aspects, including: colors of blocks referring to categories found in the palette; free placement of blocks on a 2D canvas; and some specific

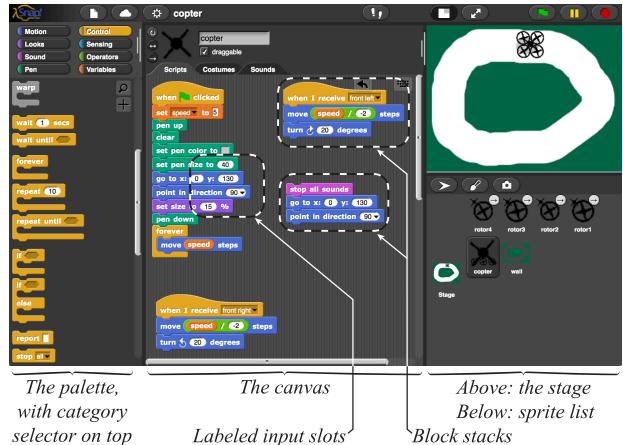


Fig. 1. The Snap! blocks-based editor. Many Blockly-based systems, including App Inventor and Default Blockly, do not have the rightmost column which includes the stage where the program is shown as it runs.

drag-and-drop behavior. As such, these families represent a ‘canonical’ style of blocks-based interfaces, which is used by many languages, including StarLogo Nova [20], Tynker [21], and ModKit [22]. In this paper, we will use terms like ‘blocks-based environments’ or ‘blocks-based editors’ to refer to the editing environments of the Scratch and Blockly families. Within the large and varied Blockly family, we restrict ourselves to (1) App Inventor, and (2) the most elaborate language that is part of the Blockly toolkit itself, which we will call *Default Blockly*. Default Blockly is demoed on the Blockly homepage [23], and part of the download, both as part of a demo called ‘code’ and as a test called ‘playground’.

III. USER STUDY

We conduct an exploratory user study that, like this analysis, is aimed at discovering what changes professionals might require in blocks-based editors. Because observing professionals using blocks-based tools, in a professional setting, is not feasible—we know of no professionals who use Scratch, Snap! or Blockly for their work—we recruit 10 4th-year students in a 4-year bachelor’s program in interaction design. Because of their 3,5 years of design-study, with close to 50% project work emulating professional work situations, and including a 5-month in-company internship, we consider them to be sufficiently representative of one group of professional end-user programmers.

We observe the participants (9 male, 1 female, ages: 20–26, average age: 22.7) perform two programming tasks in the Blockly-based language Ardublockly; conduct gaze augmented retrospective think-aloud interviews [24] with them; and have them fill out section 4 of the Cognitive Dimensions Questionnaire.

To simulate the situation of a somewhat experienced professional, working on a somewhat complex program, with a routine level of difficulty, we take four measures:

1. *Programming experience.* We select participants who have just completed, successfully, a 6-week programming course (JavaScript and Arduino-C). This gives them enough programming skills to be able to deal with programs of some size and complexity while still able to pay attention to the user experience. We do not select participants with *a lot* of programming experience, because a deep familiarity with text-based programming

might distort their response to a new, alternative programming UI.

2. *Familiar language.* We use Ardublockly [25], a blocks-based language for programming Arduinos because the participants had just completed a course in C programming with Arduinos. The editor layer of Ardublockly is the same as Default Blockly without some recent enhancements. We add some features from the C language (arrays, type declarations, parameters, local variables), that are needed for the tasks, to Ardublockly. This increases the resemblance between Ardublockly and the language the participants are familiar with, but this intervention is only on the language layer. None of our modifications change any aspect of the editor layer [26].
3. *Familiar programming problem.* Participants are given two programming tasks asking them to simulate a slot-machine on an Arduino, including the rolling of the reels, interaction through buttons and sliders, and implementing rules about winning/losing credits, payouts, raising the bet amount, etc. These tasks are taken from an assignment that these students have done, successfully, one or two weeks earlier as part of the course. This is done to prevent the cognitive load of the programming puzzle to dominate the perception of the user experience.
4. *Moderately large program.* The slot machine assignment allows us to provide the participants with a Blockly version of the program that is about 30% done. That version (Fig. 2) consists of 177 blocks including procedure definitions, arrays, global and local variables, loops, if-statements and input/output commands. The first task is to add some features to the given program, and the second task is to refactor a different, quite bad, version of the same program into one with better structure and legibility, without adding features.

Task performance takes 40 minutes per task, and the retrospective interview after each task performance is also around 40 minutes long. The participants receive no instruction concerning the CDN framework. The tasks are designed to address three of the five *user activities* from the CDN (incrementation, modification, search), but the retrospective interviews are exploratory and informal, and are therefore not structured to solicit responses directly related to the dimensions. The main results of this part of the study have been described in [2]. During the analysis of the interviews, we conclude that coding the interviews with respect to the CN framework would require too much interpretation. The most important contribution from the user study for this paper, therefore, comes from the CD Questionnaire that participants fill out after their sessions. Of the questionnaire published in [12], we use only the largest section, section 4, with questions addressing each dimension. For each dimension, we ask the students to provide two answers: one for the editor layer, and one for the language layer. This change to the questionnaire is made to increase the chances of receiving an answer about the editor layer. Section 1 is skipped because its questions are not relevant (e.g. “How long have you been using the system”, when none of the participants have ever used a blocks-based programming tool). Sections 2, 3 and 5, about user activities and sub-devices, are left out because having participants learn about, and describe sub-devices is not a very useful use of their time: Ardublockly has two features that can count as a sub-device: First, a pop-up panel for changing the structure of blocks, (like adding an else-branch to an if-block), but its

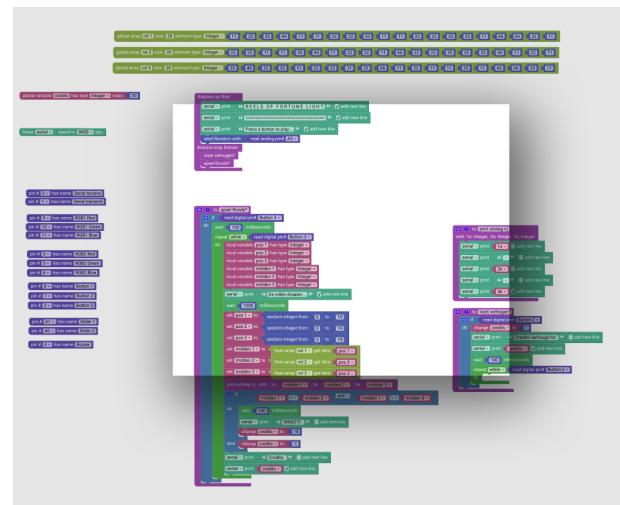


Fig. 2. Overview of the Ardublockly program that study participants were asked to extend as their first task. The highlighted rectangle shows the size of the visible area in the editor on the 1080p screen that was used. The program used in task 2 was of similar size.

notation is the same as the main notation (connecting blocks). Secondly, like Default Blockly, Ardublockly has a panel that displays the text-code it generates from the blocks, but we instruct participants not to use it, except to verify the semantics of blocks. The final question in section 4, which asks respondents to provide improvement suggestions, is given extra emphasis by making it a separate section and adding an invitation to add multiple suggestions. We translate the CD Questionnaire to Dutch and have the translation reviewed by colleagues.

IV. ANALYSIS PER DIMENSION

A. Role-Expressiveness

In [27], role-expressiveness is “intended to describe how easy it is to answer the question ‘what is this bit for?’”. The ‘bit’ can be an individual entity in the notation, or a substructure in the program, consisting of multiple entities. The ‘role’ is the intended purpose of that ‘bit’ in relation to that program. When *role-expressiveness* is low, changing a program is more likely to have unintended consequences. When *role-expressiveness* is high, it is easy to tell if the program is likely to do what the author intended to do.

RQ1: Editor properties affecting role-expressiveness: The labels of input slots add to the *role-expressiveness* of both the slot, and the block as a whole. The wording of these labels is, however, decided by the language designer, and will describe the functionality, or effect, of a block in general terms, and therefore only hint at the specific purpose for which the programmer selected it.

Block shapes indicate a syntactical category, even if they contain multiple other blocks. For example, the left-pointing puzzle-connector in Blockly expresses that a value is being calculated by the construction inside the block. Another example is the hexagonal shape of predicate blocks in Scratch and Snap!. This shape communicates to the reader that this ‘bit’ is for making a decision, even if it is being assigned to a variable instead of nested in an if-block.

With regard to inferring the purpose of larger substructures, [28] speculates that a rich set of keywords, and

the use of color, can increase *role-expressiveness* by providing ‘beacons’: quickly recognizable instructions whose presence acts as an indicator for the purpose of the surrounding code. This requires a rich, high-level set of such instructions. The discoverability of blocks, afforded by the palette, allows language designers to create quite rich vocabularies of blocks without undue damage to learnability or memorability. For this reason, the palette can be indirectly helpful for *role-expressiveness*. More direct support, however, is limited. None of the editors under consideration, for example, provide a way to subdivide larger sets of blocks into functionally related groups. GP does provide a comment-block that can sit between other blocks and could be used as a kind of sub-heading.

Defining variables and procedures allows users to create their own identifiers for describing the purpose of expressions or groups of statements, and thereby increase *role-expressiveness*. Naming things, however, is difficult, but blocks-based editors provide some features that might help with naming: (1) creating long names does not incur a penalty in having to type those long names whenever the variable is used or the procedure is called; (2) special characters like spaces, punctuation marks etc. can be used in names; (3) changing a name is very easy because the editor will change all occurrences of the name accordingly.

RQ2: Results from user study regarding role-expressiveness. Eight participants did not give answers to the CDN questionnaire that we could relate to *role-expressiveness*. The two remaining participants complained about the labelling in some of the blocks. This makes *role-expressiveness* one of the three dimensions that were least understood from the questionnaire. This is not surprising, given how Green et al. describe, in [29], how *role-expressiveness* is hard to understand, and often confused with *closeness of mapping*, one of the other dimensions for which the questionnaire yielded very few relevant answers.

The second task of the user study was to refactor a given program with much-repeated code inside a very large procedure definition. That program contained comments on multiple blocks that described the purpose of parts of the code. Although all participants were instructed, before the session, about the commenting feature, only two participants looked at those comments, which are hidden by default in Blockly.

RQ3: What design manoeuvres could improve the user interface of blocks-based editors with regard to role-expressiveness? If a rich set of blocks helps to provide beacons, editor support for large languages will help. App Inventor, for example, has made the categories in the palette dynamic, hierachic and searchable to support one of the largest feature sets in any blocks-based language. Features for *secondary notation* like commenting and layout could be improved to allow for better grouping and sub-headings within groups of blocks. Another way for editors to help users express the role of parts of the program is to decrease the barrier to introducing variables and procedures. All blocks-based languages, for example, require variables to be declared before they can be used to give a name to the result of an expression. If decreasing the *viscosity* of naming things encourages users to do so, the *role-expressiveness* in the program wins when this requirement is removed.

Expressive labels in blocks are an aspect of language design, but the editor design can help: verbose labeling could, by increasing *diffuseness*, make beacons less recognizable. Making one or two keywords in the block visually distinctive could help users discover known patterns of block combinations by focusing attention on those keywords.

B. Diffuseness

Defined in [11] as “*verbosity of language*” and in [27] as “*How many symbols or graphic entities are required to express a meaning?*”. When *diffuseness* is high, it takes more effort to scan code, and mental effort is needed to separate signal from (perceived) noise. In [11] there is speculation that verbose language may tax working memory more than compact text. When *diffuseness* is very low, on the other hand, error rates increase [11], and wholly different programs can start looking similar [27].

RQ1: Editor properties affecting diffuseness: Two properties contribute to *diffuseness* in blocks-based editors: (1) The labels for input slots require space, and also increase the amount of meaningful content to process for someone scanning the code. (2) The room required for showing block-shapes and UI affordances (borders, padding, puzzle notches, drop-down arrows, etc.) increase the screen space required for language constructs. On the other hand, a single block often represents code that requires many more lexemes (e.g. delimiters, brackets) whose placement carry meaning in a similar text-based language. This *chunking* [30] lowers *diffuseness*.

The visual design of blocks is helpful when scanning code in two ways: (1) The color differences between blocks help to see the nesting structure. Snap! and GP even alternate tints of the base color when blocks with the same color are adjacent, improving the scannability of nested loops and if-statements, where structures are nested, but the colors are identical. (2) Input slots stand out from the fixed block text, helping the user find blocks with particular parameter values.

RQ2: Results from user study regarding diffuseness: 6 (out of 10) participants responded that blocks take up more space. Some specific issues about screen space were mentioned: Blocks can grow very wide when expressions are used in input slots. The use of blocks for numeric literals (specific to Blockly) compounds this problem. Three participants remarked that function definitions with many blocks become difficult to survey quickly. Solutions were also proposed: allowing white space or comments between blocks and allowing small command blocks (i.e. statements) to be placed next to each other. Participants did not mention the visual distinctions between blocks (color, borders, etc.), and between labels and inputs, as helpful when scanning code, but only one participant described the abundance of color as distracting.

RQ3: What design manoeuvres could improve the user interface of blocks-based editors with regard to diffuseness? The authors of GP have demonstrated [31][32] a version of GP where users can drag a slider to hide or show the colors, borders, and padding that visualize block structure. At one extreme of the slider, the program looks very much like a text program. In that state, it takes about two thirds of the vertical space. This increases *visibility*, but the loss of color may hurt the discernibility of structure in a larger block stack. Within such block stacks, design manoeuvres that allow the user some control over the layout of (parts of) blocks could improve both *secondary notation* and *diffuseness*.

Removing the labels from input slots, or removing the protrusions that create specific block shapes, would substantially hurt *role-expressiveness*: users can no longer see what input slots or blocks are for, and the self-documenting aspect of blocks would be diminished. When blocks are used often, however, or when the same block is used many times, a user might find the repeating of the labels to become redundant, and experienced users might appreciate design manoeuvres

for abbreviating the labeling in blocks. Making abbreviation features a personal preference in the editor would allow users unfamiliar with the blocks to restore the labels to their original *role-expressiveness*. Editors could also show different amounts of information at different zoom levels if they were to scale the font size less than the block size. Abbreviation need not be only textual: the slider in GP, mentioned above, removes many graphical aspects around blocks, and its gradual nature seems very accommodating to different kinds of users and to different levels of experience.

C. Viscosity

The *viscosity* of a notational system describes its resistance to change. A system is more viscous when a single change (in the mind of the user) “requires an undue number of individual actions” [11]. *Viscosity* matters because end-user programmers tend to explore the requirements and design of their program while programming [5], causing frequent changes to existing code. When *viscosity* is low, the editor feels fluent and supportive of one’s thought process. When *viscosity* is high, it is cumbersome to experiment or to repair erroneous code.

RQ1: Editor properties affecting viscosity: The free placement of block stacks on the 2D canvas becomes a *viscosity* problem when such stacks are arranged close to each other. Extra space may be needed when block stacks grow, resulting in typical *knock-on viscosity* [11]: Many other stacks may need to be moved to make space for the growing block, and for each other. Blocks-based editors do not show lines connecting blocks (e.g. for data flow or control flow), so changes to the program do not suffer the additional *viscosity* of having to move blocks in order to keep the diagram intelligible.

The drag-and-drop interaction style increases *viscosity* in several ways: (1) Dropping blocks in the correct place requires attention because drop targets, such as input slots and the spots between command blocks, are small, and often near each other. (2) In none of the editors under consideration is it possible to fluently move a block to a specific place on the canvas that is not yet visible. (3) (Re)moving blocks from a sequence is quick if the user intends to also take the blocks that hang below it. Moving or removing any other subset, however, requires multiple actions in taking the structure apart, and reassembling it. (4) Creating new blocks from the palette can involve having to browse multiple categories looking for the right block. Snap!, GP and App Inventor offer a search option, but Scratch and Default Blockly do not.

In [33], the speed of editing is regarded as the key aspect which limits the use of blocks-based programming for large programs. Their solution, called *frame-based editing*, focuses on keyboard-based editing, but it also leaves out core components of the canonical blocks-based editors, such as the 2D canvas, the palette, labeled slots and more. Snap! and GP offer keyboard editing features that are promising but problematic: the user must switch to a keyboard *mode*, but leave it often because important operations, like moving a block or creating a variable, cannot be done with the keyboard.

Changing a block from one type into another type is generally not possible. In Scratch, the if-block and the if-else-block, for example, are different blocks. Blockly has separate blocks for defining procedures with, or without, a return value. Replacing one of these for the other one is often a viscous operation: dragging in a new block, moving the content from the old block into the new block, and deleting the old block. Snap!, GP and Blockly all have limited features for modifying blocks that already exist on the canvas, to add e.g. an else-

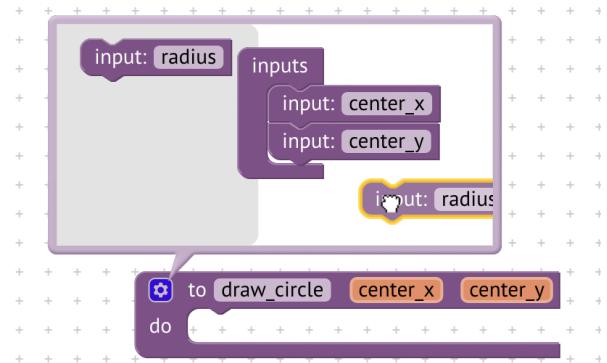


Fig. 3. The Blockly mutator panel—here shown in App Inventor to change the parameters for a procedure. Clicking the blue ‘gear’ icon toggles the panel shown on top, which is a miniature blocks-editor: a palette on the left, from which input-blocks (parameters) can be dragged into the container block on the right. It is not possible to change the procedure into one that can return a value.

branch to an if-statement. Snap! also offers a ‘relabel...’ option for changing a block into a different type: an if-block, for example, can be changed into a repeat-until-block. These specific transformation options have to be enabled by the language designer, and many that could be sensible to users are not available (e.g. wait-until \Leftrightarrow repeat-until in Snap!, or between user-defined procedures). Such transformations can only be achieved with block replacement.

As with other structure editors, programs in blocks-based editors always have a well-formed structure. This helps, in two ways, to lower *viscosity*: First, the user can rearrange constructions without having to manage the delimiters, quotes, and brackets that text-based languages use to denote structure. Second, some refactorings, such as renaming variables and functions, and adding parameters to functions, have very low *viscosity* as the editor updates all references automatically.

RQ2: Results from user study regarding viscosity: With 24 remarks, *viscosity* is the most commented-on dimension in the questionnaire. 12 remarks were positive, and 12 were negative. On the positive side, most (7) remarks relate to structure editing: manipulating syntactically complete units. Three other remarks praise the automatic updating of references.

Half (6) of the negative remarks are about the difficulty of rearranging blocks due to the fact that dragging a block out of a sequence of blocks will drag all blocks below with it. These remarks are often accompanied by a wish to be able to select multiple blocks, and then be able to drag *only* those blocks out of the sequence. The other theme in the negative remarks on *viscosity* is the need for transforming blocks into other blocks: The difficulty of adding a return value to a procedure definition is mentioned two times in relation to *viscosity*, and four times more in the rest of the questionnaire. Similarly, the ‘mutator’ panel offered by Blockly for modifying blocks (see Fig. 3) is mentioned 6 times as difficult to understand. So, the need for changing/transforming blocks seems to be there, but Blockly’s solution was not very much appreciated.

One observation from the test performances, which is supported by 5 remarks on the questionnaire, is that many participants started creating new blocks by duplicating blocks that were already in their programs. When asked to explain this, they described duplicating existing blocks as much quicker than dragging blocks from the palette.

We expected, given our participants’ experience of about 6 weeks with text-based programming, to see more remarks

about the lack of keyboard-based editing in Ardublockly, but we found none. It must be noted, however, that these participants are interaction designers and they are used to design tools like Adobe Illustrator, which are also mouse-based direct manipulation interfaces. End-user programmers from other professions, e.g. system operators or data journalists, may consider keyboard-based editing more important.

RQ3: What design manoeuvres could improve the user interface of blocks-based editors with regard to viscosity?

From the user study, we learn that the *viscosity* of some frequent drag-and-drop operations could be reduced if users were to be allowed to select just the blocks they want to (re)move, without dragging the rest of the sequence of blocks along. This will matter more when editing large block stacks. We observed participants removing a block from the middle of a large construction, with several levels of nesting. They then had trouble to reassemble the broken sequence, because the two halves were both large, and there was no indication where the detached part had originally come from.

We agree with [33], that keyboard-based manipulation of blocks could help power users increase their editing speed, but making this both intuitive and quick is an unsolved problem.

Support for modifying blocks that already exist in the program should be more intuitive than Blockly's mutators, and more flexible than Snap!'s 'relabel' feature. Being structure editors, blocks-based editors could significantly reduce the *viscosity* of larger scale changes by offering other refactorings, besides e.g. renaming, such as 'extract method' [34].

Reducing the *viscosity* of having to rearrange block stacks on the canvas when room is needed, risks diminishing an important source of *secondary notation*: Any automatic method of moving blocks will risk breaking spatial relations of block stacks on the canvas. For professional end-user programmers, this may be a price worth paying. The worst of the trade-off between layout *viscosity* and spatial *secondary notation* might be mitigated by alternative canvas designs, such as Code Bubbles [35], at the cost of complicating the user interface.

D. Secondary Notation

Secondary notation describes to what extent the user can add additional information to the program, without changing its operation or outcome. Typical channels for *secondary notation* are layout, comments, color, names and naming conventions. When *secondary notation* is high, users can easily and richly communicate about intentions, reasons and other aspects that cannot be conveyed with the language itself. *Secondary notation* can also be used to visualize or add structure.

RQ1: Editor properties affecting secondary notation: All editors, except GP, allow users to attach a comment to a block. The comment is not a block and cannot be reattached to a different block (except in Snap!). The comment can be hidden, but a small indicator of the comment's existence remains visible. In Default Blockly and App Inventor, this is the only commenting feature. Scratch and Snap! also allow comments that are not attached to blocks. GP has a comment-block for comments that is visually distinctive from other blocks and can be placed between other command blocks. This kind of comment is useful for documenting a group of blocks since the comment can remain in the group even if other blocks block are deleted. It also prevents overlapping comments: in the other editors, the panels containing comments will overlap when the blocks they belong to are close to each other. Only Snap! and GP will automatically grow and shrink the comment-block to fit the comment text, reducing the *viscosity* of

commenting. None of the editors allows for sketches, rich text, tables or hyperlinks inside the comments.

Within a block stack, the user cannot use layout, such as blank lines, for *secondary notation*, because layout in block stacks is automatic. Unconnected blocks, however, can be placed anywhere on the two-dimensional canvas, allowing for *secondary notation* by clustering related scripts, procedures, event handlers etc. Unattached comments (Scratch, Snap!, GP) can be used to explain the meaning of such clusters, but there are no other options for *secondary notation* on the canvas, such as drawing sketches or marking regions.

Blocks-based programming tools do not restrict the characters that users can use in names for variables or procedures. This feature is popular among Scratch users [36], and it works well for *secondary notation*. Spaces, brackets and other punctuation can be used to add some information hierarchy into names, e.g. "tax (percent)" or "FIX ME: draw maze". In contrast to comments, *secondary notation* in names automatically propagate to all places in the program where the name is used.

RQ2: Results from user study regarding secondary notation: All participants were aware of the commenting features of Blockly, but none of them wrote comments. This is most likely due to the fact that participants knew that they, or others, were never going to work with their program again. Participants did, however, describe some barriers to the use of comments: comments are mostly hidden (3 participants); the icon for showing and hiding a comment is only visible when there already is a comment (2 participants); and the icon indicating a comment is a question mark, suggesting a system-provided help facility instead of an opportunity to create one's own content (3 participants).

Three participants expressed the desire to add white space between blocks in a block stack in order to make large function definitions easier to comprehend.

Four participants complained about the lack of structure provided by the 2D canvas. One participant wanted to be able to draw lines on the canvas and add names to (parts of) the canvas. Another participant would draw lines on the canvas.

RQ3: What design manoeuvres could improve the user interface of blocks-based editors with regard to secondary notation? The 2D canvas is already a source of *secondary notation*, but the user study indicates that some users would be helped by the possibility to use *secondary notation* on the canvas itself to convey some macro-structure in the program. Regions could be delineated and named. Allowing users to draw on the canvas could help them link the elements of the program to design diagrams that they make. Such a drawing feature should adapt the drawing when blocks are moved.

Having a comment-block like the one in GP would be an improvement over comment panels in three ways (besides the two advantages described above): (1) Users would be able to add some grouping in large sequences of blocks. (2) It would simplify the user interface by removing some UI exceptions to the blocks-based interaction style. And (3), it would make the commenting feature more discoverable. The barriers to commenting mentioned by the study participants, suggest that the "add comment" option, only available through the (right-click) context menu, was not readily discovered. This discoverability problem is the same in all editors, except for GP, where the comment-block is prominently visible in the palette.

E. Visibility

Visibility concerns how much mental effort it takes to find and see information in the program. Low *visibility* taxes

working memory and increases programming effort when searching and navigating require time and attention. High *visibility* is helpful when programmers base programming solutions on similar solutions used elsewhere, when they adapt part of a program to work with another part, or need to understand the global structure of the program. Juxtaposition, viewing related parts of the program side-by-side, is an important contributor to high *visibility*, as are search and navigation.

RQ1: Editor properties affecting visibility: In Scratch and Snap!, programs can consist of multiple *sprites*. The editors show only the blocks that belong to the sprite that the user has selected. Each sprite has, in effect, its own canvas for its own code, and a program consists of multiple canvases if it has multiple sprites. GP and App Inventor are similar: in GP, only the blocks on the canvas for the currently selected *class* are visible, and in App Inventor, only the canvas for the currently selected *screen* is shown. In none of the editors is it possible to view code on different canvases side-by-side. Procedure definitions in Snap! are an exception: in Snap!, procedure definitions are edited in their own floating window. Several of these procedure windows can be open at the same time, so their content can be visible at the same time. Only in Default Blockly is the entire program contained in a single canvas.

Within a canvas, juxtaposition is possible by dragging one part of the program next to another part and have them both in view. This, however, becomes more difficult when blocks become very wide, leaving too little room to see the two block stacks together. GP will automatically wrap wide input slots to a next line within the block, to conserve horizontal space. Default Blockly and App Inventor have a similar feature, but the user must activate it for each block separately.

In Scratch, Snap! and GP, the canvas shares screen space with a view of the running program, called *the stage*. Juxtaposition of the code and the results of the code is prioritized over *visibility* of a larger section of the canvas. Default Blockly and App Inventor have more room for the canvas. They do not display a stage next to the canvas, and their palette only needs room for the set of categories. The set of blocks within a category is, like a menu, displayed temporarily, when the user clicks on the category. This saves some space.

Effortless navigation between parts of the program contributes positively to the *visibility* dimension, and the two tools for search and navigation in all editors are scrolling and zooming. Scrolling can be done using scrollbars, or more easily, by dragging the canvas. Zooming out can be used (not in Snap!) to get an overview of all the blocks on the canvas, but the text in the blocks becomes difficult to read. Looking for a particular name on the canvas is, therefore, not feasible when a large program is brought fully into view by zooming out. Since none of the editors has a search feature for the canvas, this leaves scrolling, in both dimensions, as the only remaining option to find a particular name in a (large) program. Zooming out *can* be an effective navigation tool for users when they can recognize the part they are looking for by the colors of the blocks.

RQ2: Results from user study about visibility: ArduBlockly only provides a single canvas for the entire program, and the participants of the user study had almost no experience with programs in multiple modules or files. It is, therefore, not surprising that none of the responses to the questionnaire discussed the (lack of) juxtaposability of multiple canvases. Four participants responded that they had dragged block stacks next to each other to view them side-by-side, but they did describe some problems with that: (1) As described above, block stacks with wide blocks can only be juxtaposed by allowing them to

overlap the other block stack. (2) There is often no free space next to a block stack, so the second block stack is dropped in a place where it is overlapping other block stacks. (3) This overlapping makes blocks stacks less readable and makes manipulation more error-prone: accidentally grabbing the wrong block tears apart the wrong block stack.

Accidentally separating block stacks also occurred when users tried to scroll the canvas, by grabbing it and dragging. Two respondents described scrolling as cumbersome. Three respondents discussed zooming as a way to get access to distant parts of the program, but two of them complained about the use of buttons for zooming. Unlike the current version of Default Blockly, ArduBlockly does not support zooming using the scroll wheel on a mouse or the zoom gesture on a trackpad; it only has buttons for zooming.

Four participants described finding things on the 2d-canvas as difficult, but they did not explain why. One likely reason, discussed in several of the interviews we held after each task performance, is that one has to scroll in two dimensions to look for some part of the program. Quickly scanning code is, according to these participants, easier if one only has to scroll in one dimension.

Two participants suggested adding a feature for searching the canvas. Four participants suggested adding a feature to highlight the definition, and other usages of a selected variable or procedure (like App Inventor, Ardublockly uses blocks on the canvas for defining variables, instead of a dialog box).

RQ3: What design manoeuvres could improve the user interface of blocks-based editors with regard to visibility? Limiting the width of blocks could help placing block stacks side-by-side. It would be even better if the user does not *have to* move blocks for juxtaposition. Many editing systems, including e.g. MS Word, allow two, independently scrollable, views on the same document. Such a two-view interface could also be used to view the contents of two different canvases at the same time, such as the code behind two different Android screens in App Inventor.

Adding a search facility seems an obvious improvement for users who want to find specific parts of their code. Some participants of the user study also expressed a need for a mental map of the entire program: an answer to the question “where is everything?”, in addition to a way to answer “where is this particular thing?”. This was prompted, in the user study, by a program that was, at the default zoom level, about 1.7 times as large as the viewport, both horizontally *and* vertically. In games, and some text editors and design tools, this answer is provided by a *mini-map*: a small, clickable rendering of the entire world or document, that is placed in a corner of the screen. Such a mini-map could also help with navigation and remove a major reason for users to want to zoom-out. In [11], adding abstraction capabilities is discussed as a useful design manoeuvre to improve *visibility*, but that is a language layer choice, and therefore out of scope for this analysis. The mini-map can, however, provide some support for this design manoeuvre. The mini-map could, like a street map, keep the names of important top-level constructs (e.g. class definitions, important procedure definitions) readable, increasing the value of abstractions for the *visibility* dimension.

V. DISCUSSION

In [30] Bau et. al. suggest four reasons why professionals do not use blocks: high viscosity, low information density, search and navigation in the 2d-canvas, and lack of source control facilities. For a usability analysis, source control and

collaboration are currently out of scope, since none of the blocks-based programming tools have source control features or a collaborative editor. See [37] for a promising discussion on integrating source control and real-time collaboration with blocks-based editing. Each of the remaining three reasons in [30] is directly related to dimensions featured in this analysis. Low information density is mostly about *diffuseness*, and search and navigation are about the *visibility* dimension. In all dimensions there are possibilities for adapting the user interface to the needs of professional end-user programmers.

To be able to measure the efficacy of these design manoeuvres, we need prototypes to be tested by subjects from the target audience in a longitudinal study. Short running tests with novice programmers may measure learnability more than other characteristics of usability, but evaluations from experienced (end-user) programmers may be skewed by selection bias: experienced programmers are already comfortable with text-based programming.

Applying the design manoeuvres carries some risks to the overall user experience. First, as stated earlier, many design improvements in one dimension may reduce usability in another dimension. Second, additional editor features may clutter the interface, both visually and cognitively. This may be less of a problem for adult knowledge workers than for children in school, but learnability is equally important to end-user programmers as it is in formal education. For this reason, we would favor design manoeuvres that do not radically change, or complicate, those features of blocks-based programming that contribute most to their learnability.

Not all design challenges posed by our target audience can be revealed by a usability analysis of existing systems. In particular, support for large languages with very rich feature sets, or even multiple languages, is likely to require a rethinking of some core features of blocks-based editing: The palette, for example, will need to accommodate many more blocks, and the current design may not be able to cope. With many categories of blocks, it may no longer be useful to denote the semantic category of the block by its background color. The set of readily distinguishable colors is small, and small color differences could increase *error-proneness*. Likewise, using block shapes to denote syntactic category or datatype, may not hold up when more syntactic categories or a more complex type system must be supported.

A. Threats to validity

The user study was an exploratory study, the start of a design effort. Four threats to validity apply to the results. First, the participants are not fully representative of the target audience: they are interaction designers, not a mix of professions. In the section on *viscosity*, we discussed how this may have influenced their responses regarding keyboard-based editing.

Second, the tasks did not involve *exploratory design*. This is one of the five user activities described as part of the CDN framework, and one that is very relevant for end-user programming. Instead, the study focused on the user activities *incrementation* and *modification*. Exploratory design, however, is defined in [11] as “*combining incrementation and modification, with the further characteristic that the desired end state is not known in advance*”, so two core aspects of exploratory design are addressed by the study design.

Third, the user study evaluated only one of the editors under consideration. Our version of ArduBlockly features the Default Blockly editor, but with a subset of C as its language. Any other choice of editor would have forced the participants

to use an unfamiliar language, which we consider a larger threat to validity: our focus is on usability aspects *other than* learnability, so we did not want participants to be spending much mental resources on learning the semantics of e.g. Snap! or App Inventor during the test. Still, Default Blockly does lack some UI features that are discussed in the analysis part of this paper, such as multiple canvases (impacting *visibility*), and keyboard-based block manipulation (impacting *viscosity*).

The fourth threat to validity is that questions in the questionnaire were misunderstood by some participants. For the dimension *hard mental operations*, for example, eight respondents described difficulties in finding blocks with the desired functionality in the palette, or issues with readability. *Hard mental operations*, however, refers to high demand on cognitive resources, when, for example, a user has to puzzle the meaning of combinations of items in the program in the head. Other dimensions where less than half of the respondents gave answers that relate to the dimension are *closeness of mapping* and *role-expressiveness*. In the latter case, this was partly triggered by the translation. The translation of “*When reading the notation, is it easy to tell what each part is for in the overall scheme? Why?*” included the Dutch phrase “*past in het grotere geheel*”, but “*past*” can also be interpreted as “*fits*”. This caused 4 of the participants to focus their answers on fitting together the puzzle-shaped blocks.

VI. CONCLUSION AND FUTURE WORK

The goal of this analysis is to propose a set of design manoeuvres, grounded in an understanding of strengths and weaknesses of the current canonical blocks-based editors, but aimed at a different target audience. We analyzed five dimensions that are relevant both to the editor layer and to the usability characteristics of programming time/effort, program comprehension, and programmer comfort. For each dimension, we found that an analysis of generic editor properties, combined with user study results, did yield such a set of design manoeuvres.

According to [16], improvements in *viscosity* are likely to benefit programming time/effort and programmer comfort. Improvements in *role-expressiveness*, *visibility*, and *secondary notation* are listed as beneficial for program comprehension, but we suggest that improvements to *visibility* will also benefit programmer comfort, and probably programming time/effort. Improvements in *diffuseness* are expected to be helpful for all three usability characteristics under consideration.

Future work will consist of designing and building a prototype block-based editor that supports multiple languages and incorporates the design manoeuvres resulting from this analysis. This editor should retain, to a high degree, the properties, as listed in [30], that make the canonical blocks-based editors very learnable. We will evaluate it in a longitudinal study with professionals in the fields of interaction design and IT systems management. In this research, we will invest particular attention to *viscosity*. Both [30] and [38] argue that editing speed is a major reason why current blocks-based editors are ill-suited for professional use. We find this reflected in our user study, where *viscosity* was the dimension that received, by far, the most comments from the participants.

From the results of the analysis presented in this paper, we conclude that there is ample room for improvement *within* the canonical style of blocks-based programming to the address the usability needs of professional end-user programmers.

REFERENCES

- [1] C. Johnson and A. Abundez-Arce, "Toward Blocks-Text Parity," in 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), 2017, vol. 1, pp. 413–419.
- [2] R. Holwerda and F. Hermans, "Towards blocks-based prototyping of web applications," in 2017 IEEE Blocks and Beyond Workshop (B B), 2017, pp. 41–44.
- [3] D. Weintrop et al., "Evaluating CoBloX: A Comparative Study of Robotics Programming Environments for Adult Novices," in Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, New York, NY, USA, 2018, pp. 366:1–366:12.
- [4] A. F. Blackwell, "End-User Developers – What Are They Like?" in New Perspectives in End-User Development, Springer, Cham, 2017, pp. 121–135.
- [5] A. J. Ko et al., "The state of the art in end-user software engineering," ACM Comput. Surv. CSUR, vol. 43, no. 3, p. 21, 2011.
- [6] A. F. Blackwell et al., "Cognitive Dimensions of Notations: Design Tools for Cognitive Technology," in Cognitive Technology: Instruments of Mind, Springer, Berlin, Heidelberg, 2001, pp. 325–341.
- [7] M. Bellingham, S. Holland, and P. Mulholland, "A cognitive dimensions analysis of interaction design for algorithmic composition software," in Proceedings of Psychology of Programming Interest Group Annual Conference 2014 (Benedict du Boulay and Judith Good, eds.), 2014, pp. 135–140.
- [8] M. Kauhanen and R. Biddle, "Cognitive Dimensions of a Game Scripting Tool," in Proceedings of the 2007 Conference on Future Play, New York, NY, USA, 2007, pp. 97–104.
- [9] F. Turbak, D. Wolber, and P. Medlock-Walton, "The design of naming features in App Inventor 2," in 2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2014, pp. 129–132.
- [10] J. Nielsen and R. Molich, "Heuristic Evaluation of User Interfaces," in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, New York, NY, USA, 1990, pp. 249–256.
- [11] T. Green, T. R., and A. Blackwell, "Cognitive Dimensions of Information Artefacts: a tutorial," 01-Jan-1998. [Online]. Available: <http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf>
- [12] A. F. Blackwell and T. R. Green, "A Cognitive Dimensions questionnaire optimised for users," in proceedings of the twelfth annual meeting of the psychology of programming interest group, 2000, pp. 137–152.
- [13] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The Scratch Programming Language and Environment," ACM Trans. Comput. Educ., vol. 10, no. 4, pp. 1–15, Nov. 2010.
- [14] N. Fraser, "Ten things we've learned from Blockly," in 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond), 2015, pp. 49–50.
- [15] J. Nielsen, Usability Engineering. Elsevier, 1994.
- [16] C. D. Hundhausen, "Using end-user visualization environments to mediate conversations: a 'Communicative Dimensions' framework," J. Vis. Lang. Comput., vol. 16, no. 3, pp. 153–185, Jun. 2005.
- [17] B. Harvey and J. Möning, "Snap! 4.1 Reference Manual," 2017. [Online]. Available: <https://snap.berkeley.edu/SnapManual.pdf>.
- [18] "About · GP Blocks," GP Blocks. [Online]. Available: <https://gpblocks.org/about/>. [Accessed: 24-Apr-2018].
- [19] D. Wolber, H. Abelson, and M. Friedman, "Democratizing Computing with App Inventor," GetMobile Mob. Comp Comm, vol. 18, no. 4, pp. 53–58, Jan. 2015.
- [20] "Welcome to Starlogo Nova." [Online]. Available: <http://www.slnova.org/>. [Accessed: 24-Apr-2018].
- [21] "Coding for Kids," Tynker.com. [Online]. Available: <https://www.tynker.com>. [Accessed: 24-Apr-2018].
- [22] A. Millner and E. Baafi, "Modkit: Blending and Extending Approachable Platforms for Creating Computer Programs and Interactive Objects," in Proceedings of the 10th International Conference on Interaction Design and Children, New York, NY, USA, 2011, pp. 250–253.
- [23] "Blockly," Google Developers. [Online]. Available: <https://developers.google.com/blockly/>. [Accessed: 24-Apr-2018].
- [24] A. Hyrskykari, S. Ovaska, P. Majaranta, K.-J. Räihä, and M. Lehtinen, "Gaze path stimulation in retrospective think-aloud," J. Eye Mov. Res., vol. 2, no. 4, 2008.
- [25] Carlos Pereira Atencio, "Ardublockly," Embedded Log. [Online]. Available: <https://ardublockly.embeddedlog.com>. [Accessed: 24-Apr-2018].
- [26] R. Holwerda, "Visual programming for Arduino," 24-Apr-2018. [Online]. Available: <https://github.com/rbrtrbt/ardublockly>. [Accessed: 24-Apr-2018].
- [27] T. R. G. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework," J. Vis. Lang. Comput., vol. 7, no. 2, pp. 131–174, Jun. 1996.
- [28] T. R. G. Green, "Cognitive dimensions of notations," in People and Computers V, 1989, pp. 443–460.
- [29] T. R. G. Green, A. E. Blandford, L. Church, C. R. Roast, and S. Clarke, "Cognitive dimensions: Achievements, new directions, and open questions," J. Vis. Lang. Comput., vol. 17, no. 4, pp. 328–365, Aug. 2006.
- [30] D. Bau, J. Gray, C. Kelleher, J. Sheldon, and F. Turbak, "Learnable Programming: Blocks and Beyond," Commun. ACM, vol. 60, no. 6, pp. 72–80, May 2017.
- [31] GP Blocks, "GP Feature: Blocks to Text Slider." [Online]. Available: <https://www.youtube.com/watch?v=iXiwOppbA0>. [Accessed: 24-Apr-2018].
- [32] J. Monig, Y. Ohshima, and J. Maloney, "Blocks at your fingertips: Blurring the line between blocks and text in GP," in 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond), 2015, pp. 51–53.
- [33] N. C. C. Brown, M. Kolling, and A. Altadmi, "Position paper: Lack of keyboard support cripples block-based programming," in 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond), 2015, pp. 59–61.
- [34] P. Techapalokul and E. Tilevich, "Programming environments for blocks need first-class software refactoring support: A position paper," in 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond), 2015, pp. 109–111.
- [35] A. Bragdon et al., "Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments," in Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, New York, NY, USA, 2010, pp. 455–464.
- [36] A. Swidan, A. Serebrenik, and F. Hermans, "How do Scratch Programmers Name Variables and Procedures?," in 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2017, pp. 51–60.
- [37] D. Wendel and P. Medlock-Walton, "Thinking in blocks: Implications of using abstract syntax trees as the underlying program model," in 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond), 2015, pp. 63–66.
- [38] M. Kölling, N. C. C. Brown, and A. Altadmi, "Frame-Based Editing: Easing the Transition from Blocks to Text-Based Programming," in Proceedings of the Workshop in Primary and Secondary Computing Education, New York, NY, USA, 2015, pp. 29–38.

Stream Analytics in IoT Mashup Tools

Tanmaya Mahapatra*, Christian Prehofer, Ilias Gerostathopoulos and Ioannis Varsamidakis

Lehrstuhl für Software und Systems Engineering, Fakultät für Informatik
Technische Universität München

Email: *mahapatr@in.tum.de, prehofer@in.tum.de, gerostat@in.tum.de, ioannis.varsamidakis@tum.de

Abstract—Consumption of data streams generated from IoT devices during IoT application development is gaining prominence as the data insights are paramount for building high-impact applications. IoT mashup tools, i.e. tools that aim to reduce the development effort in the context of IoT via graphical flow-based programming, suffer from various architectural limitations which prevent the usage of data analytics as part of the application logic. Moreover, the approach of flow-based programming is not conducive for stream processing. We introduce our new mashup tool *aFlux* based on actor system with concurrent and asynchronous execution semantics to overcome the prevalent architectural limitations and support in-built user-configurable stream processing capabilities. Furthermore, parametrizing the control points of stream processing in the tool enables non-experts to use various stream processing styles and deal with the subtle nuances of stream processing effortlessly. We validate the effectiveness of parametrization in a real-time traffic use case.

Index Terms—Internet of Things, IoT mashup tools, graphical flows, end-users, stream analytics

I. INTRODUCTION

With the proliferation of ubiquitous connected physical objects commonly known as the Internet of Things (IoT) there has been a steady increase in the amount of data generated. Data analysis can help us e.g. understand the mobility pattern of users in a city or monitor the city continuously for potential traffic congestion. Despite this great potential, deriving insights from data has been typically a separate process of using Big Data analytics tools while the application development is concerned mainly with the creation of user application for relevant business use cases [1].

For IoT applications, mashups have been proposed as a way to simplify the application development. Mashup tools [2], [3] are graphical tools designed for quick software development. They typically offer graphical interfaces for specifying the data flow between sensors, actuators, and services. They offer a data flow-based programming paradigm where programs form a directed graph with “black-box” nodes which exchange data along connected arcs.

The overall problem we address here is the lack of integrated tools for both IoT development and stream analytics [1], [4]. For instance, Node-RED [5], [6] is a visual programming environment developed by IBM which supports the creation of mashups. It is however not designed for developing stream analytics applications, and, for instance, the IBM cloud solu-

tion (<https://www.ibm.com/cloud/>) features separate graphical tools for modelling data processing and analysis.

The main contribution of this paper is to propose a novel tool concept to integrate IoT mashups and scalable stream processing, based on the actor model. We show that several new concepts to control synchronous versus asynchronous communication and parallelism are important for this. Furthermore, we show that several parameters, such as the window type and size, impact the effectiveness of stream analytics. Importantly, such parameters impact not only the performance of stream processing (e.g. whether data of certain size can be processed within a specific time bound), but also determine the functional behaviour of the system (e.g. whether the logic that is based on stream analytics is effective or not).

To evaluate our proposal, we have implemented a new Java-based tool called *aFlux*. *aFlux* supports concurrent and asynchronous execution of components in mashup flows, overcoming many limitations of current mashup tools such as Node-RED. It also has built-in support for stream analytics and allows users to tune the important parameters of stream processing in the tool front-end. This simplifies the task of devising and comparing different configurations of stream processing for the application at hand. We have evaluated the practicality of the built-in parametric stream processing in *aFlux* via realistic use cases in real-time traffic control of highways.

II. aFlux CONCEPTS AND DESIGN

In this section, we present the main concepts of our new tool approach. Despite promised benefits in having data analytics in graphical mashup tool, there are several limitations of current approaches [1], [4]. First, existing tools allow users to design data flows which have synchronous execution semantics. This can be a major obstacle since a data analytics job defined within a mashup flow may consume great amount of time causing other components to starve or get executed after a long waiting time. Hence, asynchronous execution patterns are important in order for a mashup logic to invoke an analytics job (encapsulated in a mashup component) and continue to execute the next components in the flow. In this case, the result of the analytics job, potentially computed on a third party system, should be communicated back to the mashup logic asynchronously. Second, mashup tools restrict users in creating single-threaded applications which are generally not sufficient to model complex repetitive jobs. Third, mashup

tools use visual notations for the program logic which is not very expressive to model logic of complex analytics jobs.

To summarize, we designed our mashup tool, called aFlux, to support the following requirements:

- 1) asynchronous execution of components in flows;
- 2) concurrent, multi-threaded execution of components in flows;
- 3) support for modelling complex flows via flow hierarchies (sub-flows).

In designing aFlux, we decided to go with the actor model [7], [8], a paradigm well suited for building massively parallel [9], [10], distributed and concurrent systems [11], [12]. In the actor model, an actor is an agent, analogous to a process or thread, which does the actual work. Actors respond to messages, which is the only way of interaction between actors. In response to a message, an actor may change its internal state, perform some computation, fork new actors or send messages to other actors. This makes it a unit of static encapsulation as well as concurrency [13]. Message passing between actors happens asynchronously. Every actor has a mailbox where the received messages are queued. An actor processes a single message from the mailbox at any given time i.e. synchronously. During the processing of a message, other messages may queue up in the mailbox. A collection of actors, together with their mailboxes and configuration parameters, is often termed an *actor system*.

aFlux is a web-based tool. Its front-end, implemented using the React JavaScript framework, allows users to create mashup flows via a graphical editor. The main intuition is that when a user designs a flow, we model this flow in the back-end in terms of actors making an actor the basic execution unit of our mashup tool. In the implementation of aFlux back-end we have used Akka [14], a popular library for building actor systems in Java and Scala.

A mashup flow in aFlux is called *flux*. Every time a flux is saved in the front-end, its specification is sent to the back-end where it is parsed in order to create a corresponding graph model—the *Flux Execution Model*. The parser scans for special *start nodes* in the specification of a flux. Start nodes correspond to specialized actors which can be triggered without receiving any message. All other nodes correspond to normal actors which react to messages. On detection of a start node, the graph model is built by simply traversing the connection links between the nodes as designed by the user on the front-end. On deployment of a flux, a runner fetches the flux execution model and proceeds to:

- 1) Identify the relevant actors present in the graph.
- 2) Instantiate an actor system with the identified actors.
- 3) Trigger the start nodes by sending a signal.

After this, the execution follows the edges of the graph model i.e. the start actors upon completion send messages to the next actors in the graph, which execute and send messages to the next actors and so on.

A. Asynchronous Execution of Components

Components within aFlux are of two types: *synchronous* components and *asynchronous-capable* components. Synchronous components block the execution flow, i.e. when they receive a message on their input port they start execution and pass the message through their output ports upon completion. On the other hand, asynchronous-capable components have two different types of output ports, blocking and non-blocking (Figure 1). When these components receive a message on their input port, they immediately send a message via the non-blocking port (at most one per component) so that components connected to it (i.e. components that do not require the computation result of the active component) can start their execution. When the component finishes its execution, it sends messages via its blocking ports; components connected to these ports can then start their execution. This non-blocking execution paradigm helps asynchronously execute time-consuming parts of the mashup flow while ensuring other components do not get starved from execution for a longer time period.

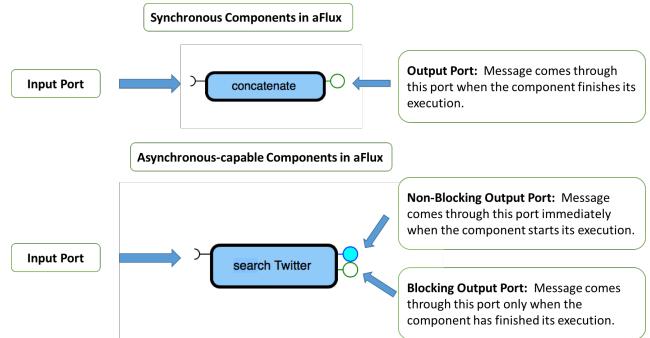


Fig. 1. Executable Components in aFlux

B. Concurrent Execution of Components

Every component in aFlux has a special configurable concurrency parameter. If a component has concurrency level of n , the actor system can spawn up to n instances of that component to process the messages concurrently. Beyond that, messages are queued as usual and processed whenever any instance finishes its current execution.

C. Sub-flows in aFlux

To encapsulate independent and reusable logic within an application flow, aFlux supports logical structuring units called *sub-flows*. A sub-flow encompasses a complete business logic and is independent from other parts of the mashup. A good candidate for a sub-flow is for example a reusable data analytics logic which involves specifying how the data should be loaded and processed and what results should be extracted. Sub-flows are modelled as asynchronous-capable components, i.e. they have input ports and two sets of output ports (i.e. blocking and non-blocking).

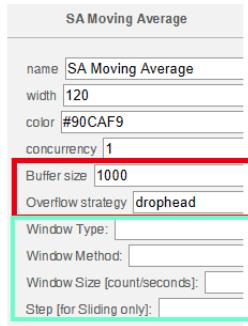


Fig. 2. Specification of buffer size, overflow strategy & window parameters.

III. STREAM PROCESSING IN AFLUX

The flow based structure of mashup tools i.e. passage of control to the succeeding component after completion of execution of the current component is very different from the requirements of stream processing where the component fetching real-time data (aka the listener component) cannot finish its execution. It must listen continuously to the arrival of new datasets and pass them to the succeeding component for analysis. Also, the listener component has many behavioural configurations which decide when and how to send datasets to the succeeding component for analysis.

In aFlux, we have introduced an abstraction called *streaming component* to model components which need to process streaming data. The implementation of streaming components relies on the Akka Streams library.

Each streaming component in aFlux offers a different stream analytics functionality (e.g. filter, merge) and can be connected to other stream analytics components or to any common aFlux component. Streaming components are categorized into *fan-in*, *fan-out* and *processing* components. Fan-in operations allow joining multiple streams into a single output stream. They accept two or more inputs and give one output. Fan-out operations allow splitting the stream into sub-streams. They accept one stream and can give multiple outputs. Processing operations accept one stream as an input and transform it accordingly. They then output the modified stream which may be processed further by another processing component.

Every stream analytics component offers a number of configurable attributes (Figure 2). The internal source of every stream analytics component has a queue (buffer), the size of which can be defined by the user (default is 1000 messages). The queue is used to temporarily store the messages (elements) that the components receives from its previous component in the aFlux flow while they are waiting to get processed. Along with the queue size, the user may also define an overflow strategy that is applied when the queue size exceeds the specified limit. It can be configured as: (i) *drop buffer*: drops all buffered elements to make space for the new element, (ii) *drop head*: drops the oldest element from the buffer, (iii) *drop tail*: drops the newest element from the buffer, (iv) *drop new*: drops the new incoming element.

TABLE I
STREAM ANALYTICS METHOD CHARACTERISTICS

Method	Responsiveness	Settling Time	Stability
Tumbling window 50	very fast	very long	very low
Tumbling window 300	slow	very short	high
Tumbling window 500	slow	none	very high
Sliding window 500	slow	none	very high

The user can also specify different windowing properties. Our implementation currently supports *content-based* and *time-based* windows. For both of these types of windows, the user can also specify a windowing method (*tumbling* or *sliding*) and also define a window size (in *elements* or *seconds*) and a sliding step (in *elements* or *seconds*) (this attribute only applies to sliding windows).

IV. EVALUATION

In order to evaluate the built-in stream processing capabilities of aFlux, we implemented an aFlux flow which involves stream processing to derive actionable insights. In our tested, the parameters of stream processing influence the end result. By selecting different values for such parameters and running different micro-benchmarks, we showcase the ease with which stream processing can be customized in aFlux and compare the different versions of the application.

Our test-bed is a traffic simulation of a highway¹ implemented in Python on top of traCI, a Python interface for SUMO microscopic traffic simulator [15]. In the scenario, a number of cars run on the highway which consists of three lanes. The cars pass over loop detectors placed next to each other at a particular mile of the highway. A loop detector measures the occupancy rate of the lane in the range of 0 to 100. A high occupancy rate signals a more busy lane and therefore the possibility of a traffic congestion. The highway operators have implemented a simple logic for reacting to traffic congestions in our simulation: if the average of the occupancy rates of the three lanes exceeds an empirical threshold of 30, a fourth lane (shoulder-lane) opens to reduce congestion. On the contrary, when the average of the occupancy rates falls below 30, the shoulder-lane closes again. We have implemented the above logic in aFlux (Figure 3), using Kafka to get the loop detector data from the simulation and communicate back the action of opening/closing the shoulder lane.

We have run different micro-benchmarks to compare the average speed of cars when changing the processing parameters of the stream of loop detector data. In particular, we have used the four methods depicted in Table I. In all micro-benchmarks, we artificially induce traffic congestion by an “accident” happening on the 500th tick of the simulation which closes one of the three normal lanes for the rest of the experiment (each experiment took 5000 ticks).

¹<https://github.com/iliasger/Traffic-Simulation-A9>

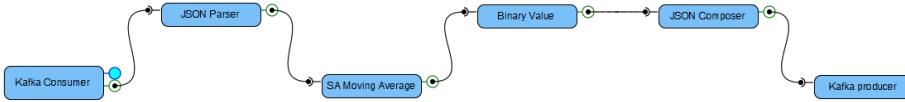


Fig. 3. aFlux flow used in the experiment - Subscribes to a Kafka topic that publishes the occupancy rates of loop detectors and calculates their moving average in real-time.

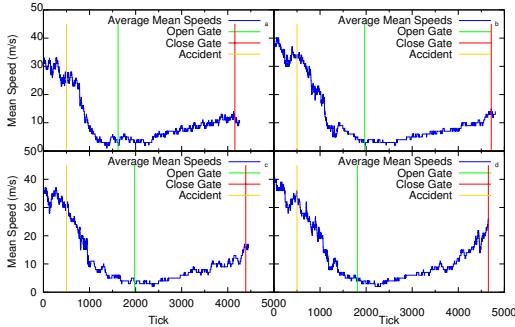


Fig. 4. Data analysis with content-based tumbling window of size (a) 50, (b) 300, (c) 500, and (d) content-based sliding window of size 500 and step 250.

The results are plotted in Figure 4 and summarized in Table I. We can observe that different methods change the time at which the extra lane is opened (*responsiveness* of the system), but they also have implications on the time when fluctuations in the shoulder-lane state end, after a change is initiated (*settling time*) and the number of fluctuations in the shoulder-lane state (*stability*). We omit the data that show the settling time and stability measurements for length constraints.

Discussion. Overall, we can make the following observations. Firstly, when data needs to be processed in real-time and the result of such analysis impacts the final outcome, i.e. performance of the application, there is no easy way to know the right stream processing method with the correct parameters. Hence, it becomes very tedious to manually write the relevant code and re-compile every time a user wants to try something new. By parametrizing the controlling aspects of stream processing it becomes easy for non-experts to test various stream processing methods to suit their application needs. Secondly, having stream processing components within aFlux allows users to quickly prototype their stream processing applications without relying on external stream processing suites. It becomes easier to prototype streaming applications, test them and finally port them to stream analytics platforms.

V. RELATED WORK

We have discussed some of the most popular mashup tools in Section I. Although these tools are good for modelling control-flow, nevertheless their in-flow data analytics capabilities are very limited [16], as discussed above. Additionally, the architecture of flow-based programming languages does not accommodate the requirements of stream processing as discussed earlier in Section II. IBM Watson Studio does not offer an integrated solution to develop IoT applications

containing in-flow data analytics [17]. One of the closest solution is Apache NiFi which is an easy to use, powerful and a reliable system to process and distribute data. It offers a highly intuitive web-based graphical user interface which allows the user to design data flows and transform data [18]. However, it does the processing via other stream processing engines (via connectors) and does not provide options to experiment with different kinds of stream processing. Kafka Streams [19], Apache Spark [20] and Apache Flink [21] are geared for developing stream processing applications however the user has to write the code using their built-in APIs and they do not have any simplified graphical user-interface for users. In addition to this, setting up and deploying clusters to try out basic stream processing increases the learning curve substantially for non-experts.

VI. CONCLUSION

In this paper, we have argued the needs for stream processing in mashup tools for IoT application development. We have demonstrated how aFlux enables rapid development of applications with stream processing integrated in the application logic which makes it unique among all the current available solutions. In this direction, the goal of the paper was to (i) integrate stream processing capabilities within aFlux (ii) parametrize the controlling factors of stream processing to the tool front-end so that it becomes easy for non-experts to try out various methods of stream processing, find the impact, tweak and re-tweak to easily arrive at the optimal configuration options for their scenario. Every stream processing component in aFlux has its own adjustable settings. This parametrization based approach makes it easy for non-experts to run adjustable stream analytics jobs. Additionally, the concurrent execution and asynchronous execution semantics of the tool facilitates non-experts to develop complex real-world applications by easy abstraction. Currently, we are working towards mapping the stream processing semantics of aFlux to popular streaming frameworks like Apache Spark and Apache Flink. This would enable non-experts to prototype streaming application using the built-in stream processing of aFlux and finally deploy the flux as a full-scale Spark or Flink application.

ACKNOWLEDGEMENT

This work is part of the TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bavarian Ministry of Economic Affairs, Energy and Technology (StMWi) through the Center Digitisation.Bavaria, an initiative of the Bavarian State Government.

REFERENCES

- [1] T. Mahapatra, I. Gerostathopoulos, and C. Prehofer, "Towards integration of big data analytics in internet of things mashup tools," in *Proceedings of the Seventh International Workshop on the Web of Things*, ser. WoT '16. New York, NY, USA: ACM, 2016, pp. 11–16. [Online]. Available: <http://doi.acm.org/10.1145/3017995.3017998>
- [2] F. Daniel and M. Matera, *Mashups: Concepts, Models and Architectures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, doi: 10.1007/978-3-642-55049-2. [Online]. Available: <http://link.springer.com/10.1007/978-3-642-55049-2>
- [3] M. Ogrin, *Mashup patterns: designs and examples for the modern enterprise*. Addison-Wesley, 2009, oCLC: ocn262433525.
- [4] "Project consortium tum living lab connected mobility: Digital mobility platforms and ecosystems," Software Engineering for Business Information Systems (sebis), München, Tech. Rep., Jul 2016. [Online]. Available: <https://mediatum.ub.tum.de/node?id=1324021>
- [5] N. Health, "How ibm's node-red is hacking together the internet of things," March 2014, <http://www.techrepublic.com/article/node-red/TechRepublic.com> [Online; posted 13-March-2014].
- [6] "IBM Node-RED, A visual tool for wiring the Internet of things." [Online]. Available: <http://nodered.org/>
- [7] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986.
- [8] C. Hewitt, "Viewing control structures as patterns of passing messages," *Artif. Intell.*, vol. 8, no. 3, pp. 323–364, Jun. 1977. [Online]. Available: [http://dx.doi.org/10.1016/0004-3702\(77\)90033-9](http://dx.doi.org/10.1016/0004-3702(77)90033-9)
- [9] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, "A foundation for actor computation," *J. Funct. Program.*, vol. 7, no. 1, pp. 1–72, Jan. 1997. [Online]. Available: <http://dx.doi.org/10.1017/S095679689700261X>
- [10] C. L. Talcott, "Composable semantic models for actor theories," *Higher-Order and Symbolic Computation*, vol. 11, no. 3, pp. 281–343, Sep 1998. [Online]. Available: <https://doi.org/10.1023/A:1010042915896>
- [11] R. Virding, C. Wikström, and M. Williams, *Concurrent Programming in ERLANG (2Nd Ed.)*. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1996.
- [12] C. Varela and G. Agha, "Programming dynamically reconfigurable open systems with salsa," *SIGPLAN Not.*, vol. 36, no. 12, pp. 20–34, Dec. 2001. [Online]. Available: <http://doi.acm.org/10.1145/583960.583964>
- [13] T. Desell, K. E. Maghraoui, and C. A. Varela, "Malleable applications for scalable high performance computing," *Cluster Computing*, vol. 10, no. 3, pp. 323–337, Sep 2007. [Online]. Available: <https://doi.org/10.1007/s10586-007-0032-9>
- [14] "Akka: Implementation of the actor model," <https://akka.io/>, accessed: 2017-12-25.
- [15] D. Krajzewicz, J. Erdmann, M. Behrisch, and L. Bieker, "Recent development and applications of SUMO - Simulation of Urban MOBility," *International Journal On Advances in Systems and Measurements*, vol. 5, no. 3&4, pp. 128–138, December 2012.
- [16] J. Mineraud, O. Mazhelis, X. Su, and S. Tarkoma, "A gap analysis of internet-of-things platforms," *CoRR*, vol. abs/1502.01181, 2015. [Online]. Available: <http://arxiv.org/abs/1502.01181>
- [17] "Put the power of AI and data to work for your business," <https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=97014197USEN>, accessed: 2018-04-20.
- [18] "Apache nifi," <https://nifi.apache.org/>, accessed: 2017-12-25.
- [19] J. Kreps, "Introducing kafka streams: Stream processing made simple," *Confluent Blog, March*, 2016.
- [20] P. Zecevic and M. Bonaci, *Spark in Action*. Manning Publications Co., 2016.
- [21] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

BONNIE: Building Online Narratives from Noteworthy Interaction Events

Vinícius Segura

IBM Research

Rio de Janeiro, RJ, Brazil
Email: vboas@br.ibm.com

Juliana Jansen Ferreira

IBM Research

Rio de Janeiro, RJ, Brazil
Email: jjansen@br.ibm.com

Simone D. J. Barbosa

PUC-Rio

Rio de Janeiro, RJ, Brazil
Email: simone@inf.puc-rio.br

Abstract—After a sensemaking process using a visual analytics application, a major challenge is to filter the essential information that led to a discovery and to communicate the findings to other people. We propose to take advantage of the interaction trace left by the exploratory data analysis, presenting it with a novel visualization to aid in this process. With the trace, the user can choose the desired noteworthy interaction steps and create a visual narrative of his/her own interaction, sharing the acquired knowledge with readers. To achieve our goal, we have developed the BONNIE (Building Online Narratives from Noteworthy Interaction Events) framework. It comprises a log model to register the interaction events and a visualization environment for users to view their own interaction history and to build their visual narratives. This paper presents our proposal for communicating discoveries in visual analytics applications, the BONNIE visualization environment, and an empirical study we conducted to evaluate our solution.

I. INTRODUCTION

Visual analytics applications (VAAps) aim to support sensemaking [1] by integrating the best of computational processing power and human cognitive prowess [2, 3, 4, 5]. A user’s interaction with a VAAp may lead to many unanticipated insights, made possible only by such combination of computational and cognitive capabilities.

After the knowledge discovery process is over, a major challenge is to recall and filter the essential information that led to the discovery and to communicate the findings to others. To share the obtained knowledge, we can wield the power of a story. Besides transmitting information, stories are a means to communicate contextual information and connect the author with the audience [6, page 19].

To leverage those capabilities, we have developed BONNIE, which allows users to interactively inspect the user interaction history log of another VAAp (for clarity, we will refer to the latter as the source system – SrcSys). The idea behind our system is to empower the SrcSys users to revisit the sequence of steps they took when interacting with the SrcSys and which led to an insight, allowing them to replicate, communicate, and share their discovery [7].

The remainder of this paper is organized as follows. Section II discusses some related work regarding annotating visualizations and data narratives. In section III, we introduce our approach to the problem and present some details of our

978-1-5386-4235-1/18/\$31.00 ©2018 IEEE

solution. Section IV details a user study that we conducted to evaluate how our approach fulfills its main goals. Finally, we conclude with section V discussing some final remarks and directions for future work.

II. RELATED WORK

Timeline visualization is commonly used to represent events in usability studies [8, 9] and system performance/usage logs¹. Our approach also involves a timeline, but it aims to empower the final user her/himself to make sense of her/his own data. The logged items in our case can be consumed by the users who interacted with the SrcSys, describing the interactions that took place in a way they can recall and communicate their discoveries about the data.

One way of documenting those discoveries about the data is by annotating visualizations. Sense.us [10] is a research prototype that focuses on visualization annotations and building tours through multiple visualization states [11]. Many Eyes² is a public website that allows the creation of dataset visualizations. Contextifier [12] is a “system that automatically produces custom, annotated visualizations of stock behavior given a news article about a company.” Many of these works focus only on a single visualization and/or do not allow to link different visualizations to create a more complex narrative. Our solution aims to integrate with VAAps, thus we need to allow annotating multiple visualizations as defined by the VAAps.

Finally, we have also researched some solutions regarding data narratives. SketchStory [13] is “a data-enabled digital whiteboard that facilitates the creation of personalized and expressive data charts quickly and easily.” Ellipsis [14] and Tableau³ support the creation of (narrative) visualizations. VisTrails [15, 16] is “an open-source scientific workflow and provenance management system that supports data exploration and visualization.”⁴ The main difference between these and our solution is the moment and context in which the user interacts with the visualization. In those works, the user interacts with the visualization “inside” the solutions themselves, defining

¹Kibana’s (<https://www.elastic.co/products/kibana>) tagline, for example, is “A Picture’s Worth a Thousand Log Lines”

²<http://www.manyeyes.com>

³<http://www.tableausoftware.com/>

⁴<http://www.vistrails.org/>

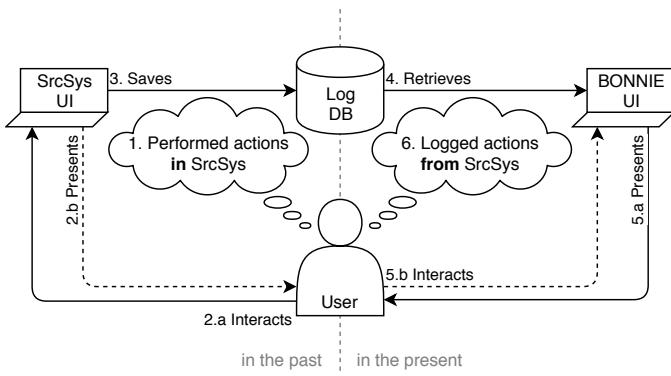


Fig. 1. Basic interaction sequence.

parameters and coordinating multiple visualizations. Our solution considers that the interaction with the visualizations happens in the SrcSys. The configuration and coordination of visualizations, therefore, is the responsibility of the SrcSys's developers. This perspective led us to explore the trace left by the interaction with the SrcSys.

Despite an extensive literature review, we could not find a solution similar to BONNIE. Many systems have an integrated history manager (*e.g.* sense.us, Tableau, VisTrails), but we have not found systems which show the history of interaction with other (instrumented) systems.

III. BONNIE

Given the importance of VAAps, there is surprisingly little support to communicate findings discovered in those applications [7, 17, 18]. Users have to rely on their own ability to document the knowledge discovery process and generate different kinds of documents to disseminate the information. As Knafllic [18, p 2] states: “being able to visualize data and tell stories with it is key to turning into *information* that can be used to drive better decision making.”

To bridge this gap, we developed BONNIE (Building Online Narratives from Noteworthy Interaction Events). It is a framework to log, revisit, and explore user interaction history from a web VAAp (the SrcSys). From the user interaction history, the user is able to recreate the visualizations from any given moment. By choosing the desired noteworthy steps, the user can create a narrative – containing visualizations and textual annotations – to document and share their discoveries or insights.

As a user interaction history visualization framework, the communication with BONNIE actually begins with the SrcSys, as shown in figure 1. The interaction sequence starts with the user interacting with the SrcSys at some point in time. During this interaction, the user has a clear goal in mind (thought balloon 1) and understanding of the interaction taking place with the SrcSys (arrows 2.a and 2.b). In the background, the SrcSys is logging this interaction in a log database (arrow 3).

Later, the same user may choose to review the interaction history, using BONNIE. BONNIE retrieves data from the log database (arrow 4) and shows it to the user. During this

interaction with BONNIE (arrows 5.a and 5.b), the user must understand the logged actions and associate them with the previous interaction (thought balloon 6), revisiting the steps s/he took in the SrcSys and electing which ones will be part of the narrative.

Figure 2 shows BONNIE’s main UI. It has two main components: a *history visualization* [19, 20, 21] (on the left) and a *narrative builder* (on the right). They work closely together so the user can visualize the interaction history and choose the relevant steps to create the desired narrative.

The history visualization showing the logged events looks somewhat similar to a Git commit graph. The most recent events are, however, at the bottom, so the history may be read in the natural direction (from top to bottom). Rows with the black squares on the graph mark inter-page navigations, showing the SrcSys, the page, and the visualizations from that page. Rows with the vertical line segments represent intra-page interactions, with a vertical line for each visualization. Each row is associated with a SrcSys action (described in textual form on the right) and displays each visualization effects that were triggered by that action (the colored nodes on the vertical lines). The SrcSys action rows may be expanded to reveal each visualization effect that was triggered in a corresponding row.

The narrative builder was created considering a slideshow/comics layout. Comics integrate images and text to communicate with an expressive and flexible language [22]. They can take small spaces to communicate complex information efficiently and effectively when compared to text-only [23], either in print or digital displays [17].

In BONNIE, the narrative is built using three main components: *panels*, *textual elements*, and *visualization elements*. *Panels* structure the narrative, creating a sequence which the readers can go through in their own pace. *Textual elements* contain text defined by the narrative author. A *visualization element* represents a given *visualization component* at a given time of the interaction.

After the user creates a narrative (by defining the *panels* and *elements*), s/he can save it. This creates a link to a web page in which any reader can view and interact with the narrative. When the narrative is displayed, each *panel* occupies the whole available screen space. The reader can scroll through different *panels* to read the whole narrative in a linear fashion. This web page has dynamic *visualization components*, meaning that the reader may interact with the *visualization components* (*e.g.* showing tooltips with data values, as when interacting with the SrcSys).

IV. USER STUDY

To evaluate BONNIE, we used WISE (Weather InSights Environment) [24, 25] as the SrcSys in our study. WISE shows weather-related data focusing on data from a given *forecast* and comparing it to the real observed data. By showing observed data alongside forecast data, WISE allows not only data exploration – detecting patterns and trends for forecast events – but also data verification and validation – comparing the forecast with observed data.

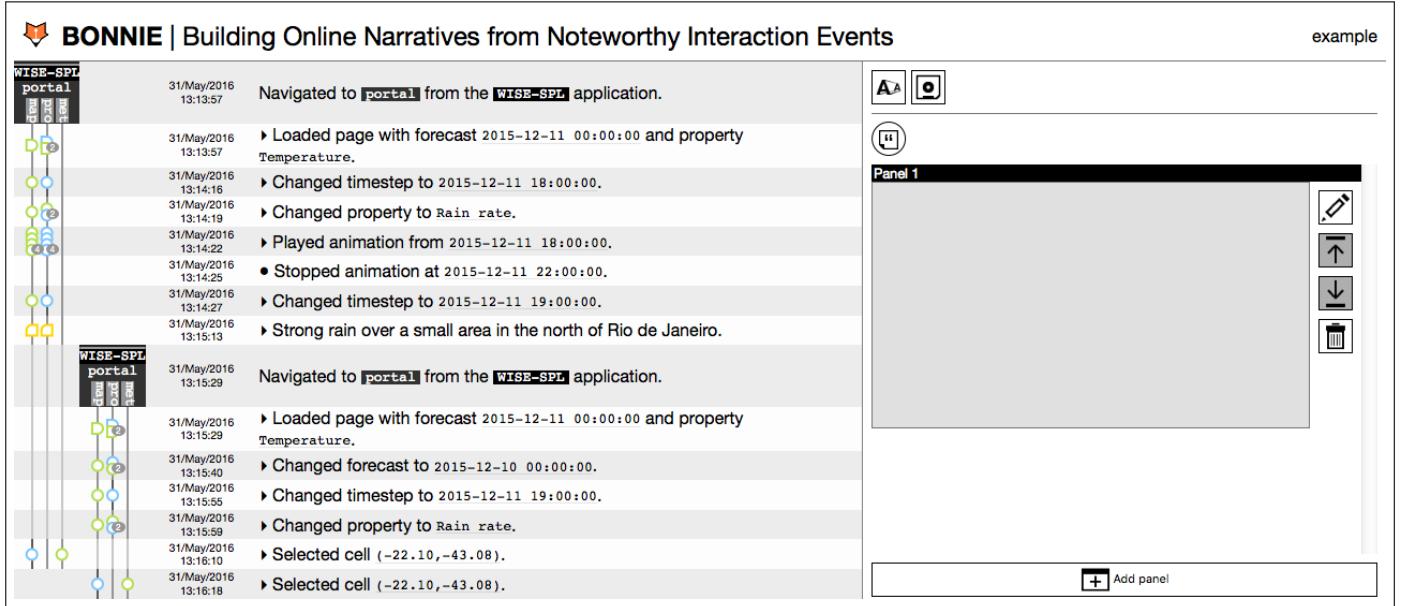


Fig. 2. BONNIE's main UI.

WISE has a fixed set of visualization components coordinated amongst themselves: (i) a *map*, displaying forecast and observed data; (ii) an *event profile*, a summary of the categorical distribution of the rain rate through the duration of the forecast; and (iii) *meteograms*, a series of line charts displaying the evolution of several weather properties over time for the selected cell in the map.

We conducted a user study with the goal of investigating how users would interact with BONNIE to create a data story based on the visualization of past interaction events. In the next section (section IV-A) we present the methodology used in the user study and in section IV-B we present some study results.

A. Study Procedure

We conducted a study with five participants, all professionals, working with software development, but not familiar with BONNIE nor WISE. The study comprised two tasks and began with an introduction to BONNIE, followed by an introduction to WISE.

Task 1 asked participants to create a narrative by choosing some interaction steps from a pre-recorded interaction session with WISE. We presented a video, narrating it once, and let the participants watch it as many times as needed. After the participant was comfortable with the video, s/he was presented with the visualization of the corresponding interaction log in BONNIE and could no longer go back to the video. We then asked for specific interaction steps to create the narrative.

Task 2 asked participants to interact with WISE, analyzing a rain event from one forecast (observing when it happened, its intensity, and its location) and comparing it with the forecast generated the day before. After the open-ended exploration of WISE, the participants should use BONNIE to create a narrative to share their interpretation. According to the goal

of the study, we did not evaluate the created narrative, only the usage of BONNIE.

After each task, the participants answered a questionnaire based on TAM [26] and TAM2 [27]. There were 22 statements, adapted to refer to BONNIE and rephrased so every statement had a “positive” meaning if the participant agreed with the statement.

The idea of answering the questionnaire after each task was to evaluate whether the actual interaction with the SrcSys would somehow impact the interaction with BONNIE. To reduce the learning effect of performing tasks in a certain order, we randomized the order in which participants performed them (P3 and P4 performed task 2 before task 1). Consequently, the study data set is not significantly biased by the learning effect [28, p. 52]. After both tasks, we interviewed the participants to gather more details about their opinions and strategies regarding BONNIE.

B. Results

The study aimed to investigate how our tool would perform in two situations: analyzing the logs from an interaction sequence that someone else performed (task 1) and from a person’s own interaction with the SrcSys (task 2). In both tasks the participants were able to identify the interaction events sequence from the history visualization. They had, however, more difficulties to identify the “key” moments that led them to some insight, having to retrace some segments of the interaction during task 2.

A single participant used the BONNIE annotation feature whilst interacting with WISE. His narrative building, therefore, was mostly guided by his own annotations. The other participants, when reminded about this feature (or when questioned about it during the interview), stated that it would have made building the narrative easier. This indicates that, when faced

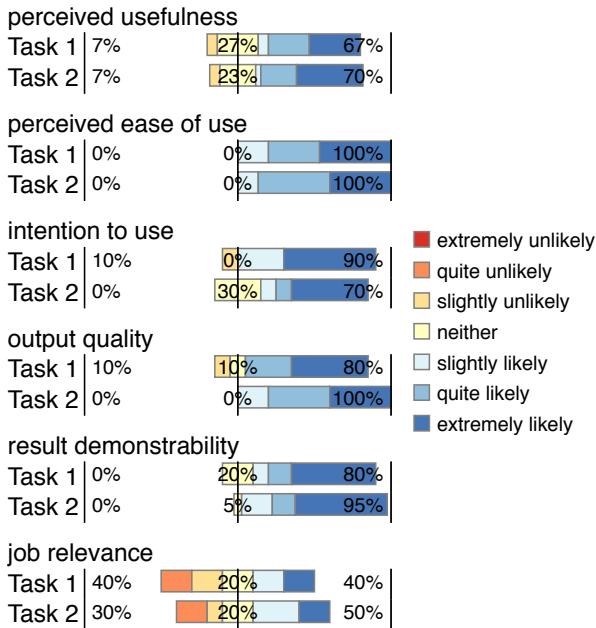


Fig. 3. Results of the user study grouped by TAM dimensions.

with the added value of BONNIE, participants were open to change their interaction with the SrcSys to make such annotations.

The questionnaire results grouped by TAM constructs can be seen in figure 3. No statistical evaluation was made rather than analyzing the answers' distribution, given our small number of participants. The overall results were positive, with most constructs having more than 50% of agreeing answers (slightly/quite/extremely likely), with the exception of "job relevance". This result was expected, because participants were not WISE's target users.

During the follow-up interview, we asked their opinion about what kind of applications would most benefit from BONNIE. All participants answered something along the lines of "applications in which you perform some analysis and must create some kind of report." This indicates that participants were able to grasp the purpose of BONNIE.

A follow-up question asked participants whether the history visualization could be useful to them. After some thought, most participants came with some kind of use case in which a system like BONNIE would be useful in their daily tasks, provided it could be integrated with a wider range of SrcSys (*i.e.* not focused only on VAAppls). This was an interesting feedback for the project and is under consideration for future development.

Observing the results for each task, we notice that the only construct which did not prove more positive for task 2 than for task 1 was "intention to use". We hypothesize that this result is closely related to the "job relevance" results, since most participants could not integrate BONNIE very well in their daily tasks since they do not interact with VAAppls as part of their jobs.

One final observation was that many participants were

not aware of their own generated content. Some participants "tested" their narratives whilst building them, while others just focused on adding content. After they considered task 2 completed, the evaluator would review the participant's narrative under the pretext of fixing the layout. During this review, many participants were surprised when what they had in mind was contrasted with their actual choices – different visualizations, different states, etc. This indicates that another interesting study would be to evaluate the generated narrative itself, both from the author's perspective (how well the narrative fits in the author's desired outcome) and the reader's perspective (how well the narrative communicates the author's idea).

V. DISCUSSION AND FINAL REMARKS

In this paper we described BONNIE, a framework to log, visualize, and generate narratives from interaction events performed in a VAAppl. The idea behind our system is to empower users to revisit the sequence of steps they took while interacting with the SrcSys and which led to an insight, allowing them to replicate, communicate, and share this discovery [7].

In this research, we started to investigate whether and how a visual representation based on users' interaction history can help users to tell data stories based on relevant events and results from their past interaction. For that, we used two perspectives of interaction history analysis. First, the participants analyzed the logs from someone else's interaction with the SrcSys. Second, the participants analyzed their own interaction with the same SrcSys. The user study results evidenced the value of our solution. Participants were able to understand the represented interaction traces and create narratives from the *history visualization*.

When using BONNIE, participants faced a challenge to identify what was noteworthy from all the available log items. One might use AI to rank the collected log data in order of importance. With such an importance model, we could highlight or subdue certain steps in the *history visualization*, making it easier for users to find the events.

Moreover, we might resort to process mining techniques to identify and encapsulate sequences of log events into higher abstractions, more closely related to the user's goals (and thus at a strategic level of interaction, closer to the user's intentions). Such techniques might also help detect similar interaction patterns, allowing to compare different interaction sequences (*e.g.* compare the interaction sequence from two analysts with the same task) or even helping the user interacting with the SrcSys by suggesting the next step in the detected interaction pattern.

ACKNOWLEDGMENT

The authors wish to thank all study's participants for their time and contributions. This work was supported in part by grants from CNPq (processes #308490/2012-6, #309828/2015-5, and #453996/2014-0).

REFERENCES

- [1] D. M. Russell, M. J. Stefk, P. Pirolli, and S. K. Card, "The Cost Structure of Sensemaking," in *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, ser. CHI '93. New York, NY, USA: ACM, 1993, pp. 269–276. [Online]. Available: <http://doi.acm.org/10.1145/169059.169209>
- [2] W. Aigner, A. Bertone, S. Miksch, C. Tominski, and H. Schumann, "Towards a conceptual framework for visual analytics of time and time-oriented data," in *Simulation Conference, 2007 Winter*, 2007, pp. 721–729.
- [3] G. Andrienko, N. Andrienko, D. Keim, A. M. MacEachren, and S. Wrobel, "Editorial: Challenging problems of geospatial visual analytics," *J. Vis. Lang. Comput.*, vol. 22, no. 4, pp. 251–256, Aug. 2011.
- [4] D. A. Keim, F. Mansmann, D. Oelke, and H. Ziegler, "Visual analytics: Combining automated discovery with interactive visualizations," in *Proceedings of the 11th International Conference on Discovery Science*, ser. DS '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 2–14.
- [5] J. Kohlhammer, D. Keim, M. Pohl, G. Santucci, and G. Andrienko, "Solving problems with visual analytics," *Procedia Computer Science*, vol. 7, pp. 117 – 120, 2011, proceedings of the 2nd European Future Technologies Conference and Exhibition 2011 (FET 11). [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S1877050911007009>
- [6] W. Quesenberry and K. Brooks, *Storytelling for User Experience - Crafting Stories for Better Design*, 1st ed. New York, NY, USA: Rosenfeld Media, Apr. 2010.
- [7] M. Elias, M.-A. Aufaure, and A. Bezerianos, "Storytelling in visual analytics tools for business intelligence," in *Human-Computer Interaction – INTERACT 2013*, P. Kotzé, G. Marsden, G. Lindgaard, J. Wesson, and M. Winckler, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 280–297.
- [8] T. Carta, F. Paternò, and V. Santana, "Support for remote usability evaluation of web mobile applications," in *Proceedings of the 29th ACM International Conference on Design of Communication*, ser. SIGDOC '11. New York, NY, USA: ACM, 2011, pp. 129–136. [Online]. Available: <http://doi.acm.org/10.1145/2038476.2038502>
- [9] F. Paternò, A. G. Schiavone, and P. Pitardi, "Timelines for mobile web usability evaluation," in *Proceedings of the International Working Conference on Advanced Visual Interfaces*, ser. AVI '16. New York, NY, USA: ACM, 2016, pp. 88–91. [Online]. Available: <http://doi.acm.org/10.1145/2909132.2909272>
- [10] J. Heer, F. B. Viégas, and M. Wattenberg, "Voyagers and voyeurs: Supporting asynchronous collaborative information visualization," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2007, pp. 1029–1038.
- [11] J. Heer and M. Agrawala, "Design considerations for collaborative visual analytics," *Information Visualization*, vol. 7, no. 1, pp. 49–62, 2008. [Online]. Available: <http://iv.iis.sagepub.com/content/7/1/49.abstract>
- [12] J. Hullman, N. Diakopoulos, and E. Adar, "Contextifier: Automatic generation of annotated stock visualizations," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '13. New York, NY, USA: ACM, 2013, pp. 2707–2716. [Online]. Available: <http://doi.acm.org/10.1145/2470654.2481374>
- [13] B. Lee, R. H. Kazi, and G. Smith, "Sketchstory: Telling more engaging stories with data through freeform sketching," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 12, pp. 2416–2425, Dec 2013.
- [14] A. Satyanarayan and J. Heer, "Authoring narrative visualizations with ellipsis," *Comput. Graph. Forum*, vol. 33, no. 3, pp. 361–370, Jun. 2014. [Online]. Available: <http://dx.doi.org/10.1111/cgf.12392>
- [15] E. Santos, L. Lins, J. P. Ahrens, J. Freire, and C. T. Silva, "Vismashup: Streamlining the creation of custom visualization applications," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 15, no. 6, pp. 1539–1546, 2009.
- [16] C. T. Silva, E. Anderson, E. Santos, and J. Freire, "Using VisTrails and provenance for teaching scientific visualization," in *Computer Graphics Forum*, vol. 30, no. 1. Wiley Online Library, 2011, pp. 75–84.
- [17] B. Bach, N. Kerracher, K. W. Hall, S. Carpendale, J. Kennedy, and N. Henry Riche, "Telling stories about dynamic networks with graph comics," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, ser. CHI '16. New York, NY, USA: ACM, 2016, pp. 3670–3682.
- [18] C. N. Knaflic, *Storytelling with Data: A Data Visualization Guide for Business Professionals*. Wiley, 2015. [Online]. Available: <https://books.google.com.br/books?id=retRCgAAQBAJ>
- [19] V. C. Segura, J. J. Ferreira, R. F. de G. Cerqueira, and S. D. J. Barbosa, "An analytical evaluation of a user interaction history visualization system using CDN and PoN," in *Proceedings of the 15th Brazilian Symposium on Human Factors in Computer Systems*, ser. IHC '16. New York, NY, USA: ACM, 2016, pp. 28:1–28:10. [Online]. Available: <http://doi.acm.org/10.1145/3033701.3033729>
- [20] V. C. V. B. Segura and S. D. J. Barbosa, *History Viewer: Displaying User Interaction History in Visual Analytics Applications*. Cham: Springer International Publishing, 2016, pp. 223–233.
- [21] V. Segura and S. D. J. Barbosa, "Historyviewer: Instrumenting a visual analytics application to support revisiting a session of interactive data analysis," *Proc. ACM Hum.-Comput. Interact.*, vol. 1, no. EICS, pp. 11:1–11:18, Jun. 2017. [Online]. Available: <http://doi.acm.org/10.1145/3095813>
- [22] S. McCloud, *Understanding Comics*. Kitchen Sink Press, 1993. [Online]. Available: <http://books.google.com.br/books?id=5aQNAQAAMAAJ>

- [23] M. J. Green and K. R. Myers, "Graphic medicine: Use of comics in medical education and patient care," *BMJ*, vol. 340, 2010. [Online]. Available: <http://www.bmjjournals.org/content/340/bmj.c863>
- [24] J. S. J. Ferreira, V. Segura, and R. Cerqueira, "Cognitive dimensions of notation tailored to environments for visualization and insights," in *Proceedings of the XIV Brazilian Symposium on Human Factors in Computer Systems*, ser. IHC 2015, 2015.
- [25] I. Oliveira, V. Segura, M. Nery, K. Mantripragada, J. P. Ramirez, and R. Cerqueira, "WISE: A web environment for visualization and insights on weather data," in *WVIS - 5th Workshop on Visual Analytics, Information Visualization and Scientific Visualization*, ser. SIBGRAPI 2014, 2014, pp. 4–7. [Online]. Available: <http://bibliotecadigital.fgv.br/dspace/bitstream/handle/10438/11954/WVIS-SIBGRAPI-2014.pdf?sequence=1>
- [26] F. D. Davis, "Perceived usefulness, perceived ease of use, and user acceptance of information technology," *MIS Quarterly*, vol. 13, no. 3, pp. 319–340, 1989. [Online]. Available: <http://www.jstor.org/stable/249008>
- [27] V. Venkatesh and F. D. Davis, "A theoretical extension of the technology acceptance model: Four longitudinal field studies," *Management Science*, vol. 46, no. 2, pp. 186–204, 2000. [Online]. Available: <http://dx.doi.org/10.1287/mnsc.46.2.186.11926>
- [28] J. Lazar, J. H. Feng, and H. Hochheiser, *Research Methods in Human-Computer Interaction*. John Wiley & Sons Ltd, 2010.

What Programming Languages Do Developers Use? A Theory of Static vs Dynamic Language Choice

Aaron Pang, Craig Anslow, James Noble

School of Engineering and Computer Science

Victoria University of Wellington, New Zealand

Email: {pangaaro, craig, kxj}@ecs.vuw.ac.nz

Abstract—We know very little about why developers do what they do. Lab studies are all very well, but often their results (e.g. that static type systems make development faster) seem contradicted by practice (e.g. developers choosing JavaScript or Python rather than Java or C#). In this paper we build a first cut of a theory of why developers do what they do with a focus on the domain of static versus dynamic programming languages. We used a qualitative research method – Grounded Theory, to interview a number of developers ($n=15$) about their experience using static and dynamic languages, and constructed a Grounded Theory of their programming language choices.

I. INTRODUCTION

With an increasingly number of programming languages, developers have a wider set of languages and tools to use. Static languages are generally considered to have inflexible code and logical structures, with changes only occurring if the developer makes them. While dynamic languages allow greater flexibility and are easier to learn. It can be difficult to select a language(s) for a given project. There is little research regarding why and how developers make the languages choices they do, and how these choices impact their work.

Over the last decade there has been an increase in the use of dynamic languages over static ones. According to a programming language survey in 2018 [1], three of the five most popular programming languages are dynamic. These are Python, JavaScript, and PHP which accounted for 39%, while the remaining two C# and Java accounted for 31%. In 2007 the top five were Java, PHP, C++, JavaScript, and C.

In this paper we investigate why and how developers choose to use dynamic languages over static languages in practice and vice versa. We created a *theory of static vs dynamic language choice* by interviewing developers ($n=15$) and using Grounded Theory [2], [3]. The theory discusses how three categories (attitudes, choices, and experience) influences how developers select languages for projects, the relationships between them and the factors within these categories. Attitudes describes the preconceptions and biases that developers may have in regard to static or dynamic languages, while choice is the thought process a developer undergoes when selecting a programming language, and experience reflects the past experiences that a developer has had with a language. The relationships between these categories was that attitudes informs choice, choice provides experience, and experience shapes attitudes.

II. RELATED WORK

A number of papers about programming languages have conducted empirical studies, controlled experiments, surveys, interviews with developers, and analysis on repositories.

Paulson [4] discusses the increase in developers using dynamic languages rather than static ones. Paulson claims that developers wish to shed unneeded complexity and outdated methodologies, and instead focus on approaches that make programming faster, simpler, and with reduced overhead.

Prechelt and Tichy [5] conducted a controlled experiment to assess the benefits of procedure argument type checking ($n=34$, with ANSI C and K&R C, type checked and non-type checked respectively). Their results indicated that type-checking increased productivity, reduced the number of interface defects, and reduced the time the defects remained throughout development. While Fischer and Hanenberg [6] compared the impact of dynamic (JavaScript) and static languages (TypeScript) and code completion within IDEs on developers. The results concluded that code completion had a small effect on programming speed, while there was a significant speed difference between TypeScript (in favour of) and JavaScript. A further study [7], compared Groovy and Java and found that Java was 50% faster due to less time spent on fixing type errors which reinforces findings from an earlier study [8]. Another study compared static and dynamic languages with a focus on development times and code quality using Purity (typed and dynamically typed versions). Results showed that the dynamic version was faster than the static version for both development time and error-fixing contradicting earlier results [9].

Pano et al. [10] interviewed developers to understand their choices of JavaScript frameworks. The theory that emerged was that framework libraries were incredibly important, frameworks should have precise documentation, cheaper frameworks were preferred, positive community relationships between developers and contributors provided trust, and that developers highly valued modular and reliable frameworks.

Meyerovich and Rabkin [11] conducted surveys to identify the factors that lead to programming languages being adopted by developers. Analysis of the surveys lead to the identification of four lines of inquiry. For the popularity and niches of languages, it was concluded that popularity falls off steeply

and flatlines according to a power law, less popular languages have a greater variation between niches and developers switch between languages mainly due to its domain and usage rather than particular language features. In terms of understanding individual decision making regarding projects, they found that existing code and expertise were the primary factors behind selecting a programming language for a project, with the availability of open-source libraries also having a part. For language acquisition, developers who had encountered certain languages while in education were more likely to learn similar languages faster while in the workforce. Developer sentiments about languages outside of projects were examined with developers tending to have their perceptions shaped by previous experience and education. Developers tended to place a greater emphasis on ease and flexibility rather than correctness, with many of those surveyed being pre-disposed to dynamic languages. They concluded that all of these factors are relevant to the adoption of programming languages.

Ray et al. [12] conducted a large study of GitHub projects ($n=729$, 17 of the most used languages and the top 50 projects for each of these), to see the impact static and dynamic languages as well as strong and weak typing have on the quality of software. Projects were split into different types and quality was analysed by counting, categorising, and identifying bugs. They concluded that static typing is better than dynamic typing and strong typing better than weak.

Prior research has not explained why developers use certain programming languages for work and personal projects and why there is an increase in the use of dynamic languages. However, the results of prior research will help inform our research as it analyses where dynamic development may be utilized rather than static development, as well as running counter to the belief that statically typed development is always faster, less error-prone, and easier to fix than dynamically typed development. The results will be in turn used in our study to identify potential avenues of questioning for data collection and analysis, allowing us to identify why developers use static or dynamic languages.

III. METHODOLOGY

We used the Grounded Theory (GT) method as it supports data collection via interviews and we were primarily concerned with the subjective knowledge and beliefs that developers hold, rather than technical ability [2], [3]. As our research focuses on people and the decisions that they make in regards to programming, GT is appropriate to study these behaviours and interactions particularly for software development projects as used elsewhere [13]–[20]. Human ethics approval was obtained. There are several stages to GT. Upon identifying a general research topic, the first step is the sampling stage, where potential participants are identified and a data collection method selected. We used interviews for data collection and transcribed each interview from audio recordings. Next is data analysis which uses a combination of open coding and selective coding. Coding is where key points within the data are collated from each interview and summarised into several

TABLE I: A summary of the participants. Participant ID, Role Type based on developer role, Experience in number of years, Organization Type, Programming Languages experience in their main top two languages.

PID	Role	Exp	Organization	Languages
P1	Graduate	1	Government	Java, JavaScript
P2	Graduate	1	Finance	C#
P3	Graduate	1	Accounting	JavaScript, C#
P4	Graduate	1	Development	Java, Python
P5	PhD Student	4	Energy	Java, Coq
P6	PhD Student	4	Education	JavaScript, Python
P7	Intermediate	>5	Consultancy	C#, JavaScript
P8	PhD Student	1	Education	Python, C++
P9	Senior	>10	Self-Employed	Python
P10	Senior	40	Consultancy	Python
P11	Senior	10	Development	C++ , Objective C
P12	Senior	>10	Development	JavaScript, TypeScript
P13	Graduate	4	Development	Java, JavaScript
P14	Intermediate	>5	Development	Closure, JavaScript
P15	Intermediate	>5	Development	Closure, JavaScript

words. These codes can be formed into concepts, which are patterns between groups of codes and then categories, which are concepts that have been grown to encompass other concepts. Amongst these categories a core category will emerge, which will become the primary focus of the study. Selective coding can then be used which only deals with the core category. Throughout the process is the memoing task, where ideas relating to codes and their relationships are recorded [21]. By recording all of these memos, it allows knowledge about what is emerging from the data and its analysis. One can then revisit the data collection phase and adjust their approach to specifically ask participants questions related to the core category. In order to create a theoretical outline, the memos are conceptually sorted and arranged to show the concept relationships. This theoretical outline will show how other categories are related to the earlier identified core category. Theoretical saturation is when data collection is completed and no new top-level categories are being generated.

The first author conducted interviews with 15 participants, see Table 1. The interview schedule was updated after the first interview in order to provide a greater depth of questioning. The initial schedule only had broad sections, whereas the revised schedule had specific questions within each section indicating potential lines of questioning. By asking more open-ended questions, we were able to attain high-quality responses that contained more information. Once emergent information became apparent, questions in future interviews were modified in order to reflect these trends. From the interviews several concepts emerged from the open codes. The concepts have been formed by identifying groups of codes that have broad similarities (some codes were in multiple concepts) which helped to find the main theme of the research. Aggregating the codes helped to inform the factors that determine why developers make the choices they do regarding utilising static or dynamic languages. The first author identified the codes and to support reliability the others validated them to decide the concepts. Further details about the interview procedure, interview data, and coding results can be found elsewhere [22].

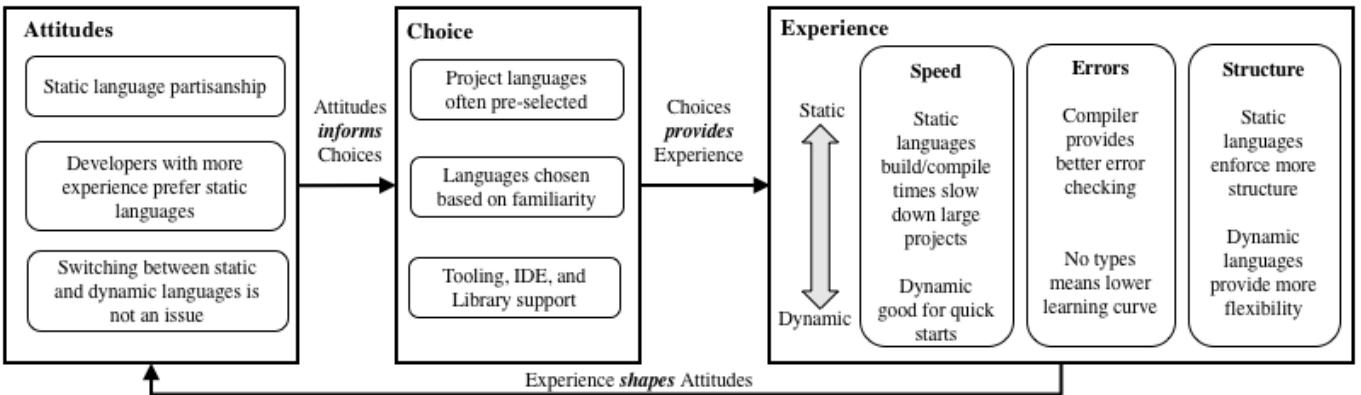


Fig. 1: Theory of static vs dynamic language choice with categories: Choice, Experience, and Attitudes, their relationships, and the factors that influence them. Choice provides experiences, experiences shapes attitudes, and attitudes informs choice.

IV. THEORY OF STATIC VS DYNAMIC LANGUAGE CHOICE

The primary theory emergent from the data is that there are several factors that underpin programming language choice, see Fig. 1. These factors can be aggregated into three key categories: *attitudes*, *experience*, and *choice*. In addition to being influenced by their factors, categories can also influence each other, with experience shaping attitudes, attitudes informing choice, and choice providing experience.

A. Attitudes

The Attitudes category represents a developer's existing bias for or against a certain language or class of language, which influences their overall decision making. If a developer has a positive bias towards a static programming language, they are more likely to use it for personal and enterprise projects and likewise if they prefer dynamic languages. This can also hold true for negative perceptions of programming languages. If a developer has a negative perception of a language, this will also impact language choice. There are several factors that can shape a developer's attitudes which include: static language partisanship, switching between static and dynamic languages not being an issue, and that more experienced developers tend to prefer static programming languages.

1) *Static language partisanship*: Refers to how developers that primarily use static languages feel strongly about the advantages they offer. Several participants indicated that they mostly used typed languages. Due to these languages performing error checking during compile time rather than at runtime, participants felt more secure about their code being error free when executed. These participants also believed that dynamic languages did not offer the same level of error checking, with errors potentially being present in programs using dynamic languages for longer periods of time.

"It gives you a better sense of security in the end that you've done something, you can leave it and it's working. If you need to touch it, the compiler will tell you why. There's a sense of security once you run the compiler and

it tells you it's ok. With JavaScript, you could have a typo and not notice it for 5 years." P7

Participants who strongly approved of static languages found that using dynamic languages was not faster and felt using a static language produced greater efficiency and programming speed. Although there was more to type due to having to declare types, participants who strongly supported static languages claimed developers should be fast typists anyway and that the additional time spent was often saved when it came to error checking and fixing bugs. It was commonly stated that dynamic languages were less reliable and that having a type-checking system allowed for better planning and more structured development due to having to consider the type of each object and how it would be utilised.

Those who used dynamic languages were less vocal in their support for static or dynamic languages. They looked at both types of languages equally and considered the merits and drawbacks for both. Participants who primarily used static languages were strongly in support of them and a few stated that they would not use dynamic languages unless there was absolutely no alternative. They frequently cited that many dynamic languages had a static counterpart that would enable them to have the benefits a static language offered (e.g. TypeScript being a static version of JavaScript).

Static language partisanship shows a clear indication of developer's bias towards static languages and against dynamic languages. For many developers, there is an ingrained inclination towards the usage of static languages to the point where dynamic languages are not considered unless absolutely necessary, which is significant in programming language choice.

2) *Developers with more experience prefer static languages*: Refers to how developers with more experience programming tended to more strongly support the usage of static languages for personal and industry projects rather than dynamic languages. One possible reason for this is that significant usage of dynamic languages has only picked up recently, while participants with more experience will have been programming for companies and projects well before this shift

towards web programming. This would imply that experienced developers have more history with static languages and may feel more comfortable using them.

"In my experience, where I have found serious problems is that say I made a typo, dynamic languages don't tell me anything at all. I don't find out until I eventually see that the code is not working and then I check that the spelling is wrong. If I had the ability to pre-declare and if I try to reference a member I didn't declare, it'd immediately throw an exception and tell me to fix it." P10

Another reason is that more experienced developers are more likely to hold senior roles within project teams, often acting as managers or lead developers. Thus, they may value different traits in programming languages than junior developers or those who focus more on personal projects and start-ups. There may be an emphasis placed on having better error checking or enforcing structure throughout development, both of which static languages provide through having a compiler and type declaration. Some developers may have pre-existing biases which can be built up over a long period of time and tend to be further up within a team or company hierarchy, either as a lead developer or a project manager. This gives them more control over projects and language selection, which may be impacted due to this factor.

3) *Switching between static and dynamic languages not an issue:* Due to the significant differences between several static and dynamic programming languages, using either for too long may cause developers to forget some of the quirks and idiosyncrasies of other programming languages and make a transition to another project difficult. This was often not the case, with participants indicating that their training both at an educational and corporate level meant that they were well-versed in several programming languages and that alternating between them did not cause an increase in errors or mistakes.

"By and large, you've got stuff on the screen to look at and I can switch reasonably well now anyhow" P9

Some participants stated that although there were minor difficulties such as re-adjusting to the usage of curly brackets and semi-colons if returning to Java from using Python, these were often short-lived once they got back into the "swing of things." Although some developers indicated that they may have a preference for working in a given language due to experience or because they enjoyed how that language worked, this did not impact on their development capabilities in other languages that they used less, but were still familiar with.

"You'll be programming in Java & then switch to Python, add a semi-colon, and think that is not right." P4

B. Choice

The Choice category represents the thought process that a developer undergoes when actually selecting a programming language to use for a project. It is a measure of the factors that influence how much say a developer feels they have when making a language selection and whether or not they have the ability to impact this selection if it is not being made

by them. Choice is effectively the final steps in a developer's rationalisation of using a given programming language over another and the impact that they perceive it will have on the development process. There are several factors that can influence choice which include: project languages often being pre-selected, languages being chosen based on familiarity and tooling, library, and IDE support for a selected language.

1) *Project languages are often pre-selected:* Several participants indicated that the choice of programming language was not their responsibility. All participants were asked whether they selected the language used in the projects they had worked on and for most this was not the case.

"It was something that the founder learned and liked. They thought it was good for solving mathematical problems and we've used it since." P14

Languages were either selected by the lead developer or management or they came onto an existing project that was already using a certain language due to large, pre-existing code bases. This restricted the choices that were available to project teams and meant that there were sometimes few viable languages to choose from. Often, these were languages that the participants already knew. However, in instances where the participant had to learn a new language, management was generally supportive and provided assistance.

Most participants tended to believe that the languages chosen by the organisations and teams they worked within were good fits for the project. Despite not being able to significantly impact the choice of development language, most developers felt that this was not important and that they were usually brought onto projects that fit their skillset anyway. However, there were some exceptions where developers believed that the project could have better met time deadlines, budget, or functional/non-functional requirements if a different programming language was used. Another reason for why programming languages were pre-selected was that the company a participant worked for often specialised in a given language and almost all of their development was done in that language.

For lead developers, project managers and those fortunate enough to be able to have a direct impact on language selection for projects, it was important to analyse how decisions were made. Often, there was less choice available than initially believed due to restrictions such as company expertise or pre-existing code bases that were to be utilised. This was interesting as it showed that both senior developers and those further down the chain of command had this lack of choice in common and was certainly a factor in how programming language decisions were reached.

2) *Languages chosen based on familiarity:* Many participants indicated that project leads and lead developers often selected programming languages that they were personally familiar with or that they felt the majority of workers within the project team would be familiar with. One reason for this stated by participants is the difficulty in attracting new workers for projects if they were developed using languages that people were unfamiliar with. By opting to use more popular languages (e.g. JavaScript, Java, or C++) it would be easier to recruit

experienced developers for projects. Another reason that fell in a similar vein was that using a language that developers were unfamiliar with would slow down development and make it more costly. This is due to the expense with having to train people in a new language and potential increase in time spent error-checking due to inexperience being more likely to introduce bugs into project code.

"There was a lot of ugly code because it was new to me and if I could go back, I'd definitely clean it up." P3

Conversely, by selecting a familiar language, lead developers felt that development would be faster and result in a higher-quality product. When it came to selecting a programming language for personal projects or projects where they were the lead, developers often opted for languages they were familiar with or languages that were similar. Developers who favoured and were used to programming with static languages were more likely to choose those as was the reverse case.

"I don't think any decisions were made about Python because of syntactical reasons. I think they chose Python because everyone knew it." P12

Languages chosen based on familiarity show how non-technical factors can be a decider in what programming language is used for a project. Often, it is not just the technical benefits and drawbacks that must be considered, but also the benefits to the team. By selecting languages that are familiar to teams, developers believe that they are increasing the odds of success by minimising any particular learning curves.

3) Tooling, IDE and library support: These represent some of the technical factors that may impact why a specific language was chosen. Tooling refers to tool support, which are development tools that can be utilised to support and complement programming languages by providing additional functionality that they do not presently have. IDE support is defined as the set of IDEs that can be used or are compatible with the selected programming language, while library support is the list of libraries and the additional services or functionality that these provide. The support provided for a language can be an influence behind a developer's choice regarding whether or not to use that particular language. Several participants felt that tool support was a major benefit when selecting a language, due to the options it added. Some stated that it simply allowed you to do more than an equivalent language without tool support.

"Static languages enables certain tool support that you can't get otherwise or that requires type inference or runtime tracing or what have you." P9

Having multiple libraries was a benefit that many participants pointed out with similar reasoning to the upsides of tool support. They felt that it provided significantly increased functionality and a wider range of options that could be utilised when programming, with claims of time-saving due to libraries being able to provide code that would otherwise take extended periods of time to figure out and develop. On the other hand, languages with little library support meant that they had little increased functionality and may not be considered.

"We chose Java because there's a library for whatever you need to do." P5

IDE support was less of a factor, with some developers indicating that although they looked for compatibility with mainstream IDEs, they often did not want to use many of the flashier options instead preferred a more simple approach.

C. Experience

The Experience category represents the previous experiences that a developer has had with a given programming language. There are three subcategories: speed, errors, and structure. Each of these has a static factor and a dynamic factor. Speed refers to how language choice has affected speed in previous projects, errors is the error checking experience that developers have had using previous languages, and structure refers to how structured the development process was when using either a static or dynamic language.

1) Speed: Static – build/compile times slow larger projects down. For participants who worked on larger projects, the build and compile times necessitated by using a static language could become cumbersome and have a negative impact on development. This was often due to having a large number of modules being used. Participants found this to be cumbersome and that the time spent waiting for a build to compile could be completely mitigated or removed through the usage of an appropriate dynamic language.

"There's thousands of modules now in the project and TypeScript has to compile and it's really slow. We're often running out of memory in some cases, which is a real problem for us." P12

If static languages can slow down larger projects due to the increased compile and build times, this may impact the decision-making rationale of a developer for another project. Their experience of a static language providing these increased waiting times may make them less likely to employ a static language for a similarly-sized project in the future.

Dynamic – good for smaller projects and quick starts. Several participants raised the idea that dynamic languages were suited for projects that were small in scale or needed a product up and running quickly. Several mentioned both Python and JavaScript as being two languages which were easy to set up and get something working quickly. Often, this was better for personal projects, where participants may not have the time to commit heavily to them and using dynamic languages would result in observable results sooner. Developers believed that this often had an impact on project success as getting the framework up and running quickly meant that work could begin faster and less time was wasted on setting up. This allows for maximum efficiency and allocating more time and resources to the software development stage rather than being bogged down in setup.

"Dynamic languages are great for small hacky things."
P9

"The setup was super fast. You just have the command line interface, the node package manager and it all just goes. The overall setup did contribute to the project." P1

2) Errors: Static – provide better error checking. One common trend amongst all participants was that type checking and the presence of a compiler generally meant that they provided better error checking for programs. Errors were caught before runtime and the compiler or IDE would inform the developer if there was an error and what had caused it. This was different from dynamic languages, where error checking does not occur until runtime.

"A lot of errors don't show up until they actually happen in JavaScript, C# is a lot clearer since the compiler will tell you if there is an error." P2

Proponents of static languages who believed it had superior error checking often stated that dynamic languages do not inform you about misspelled items and other basic items, while they claimed that static languages would pick these up immediately and point the developer to the location of the typo due to having a compiler. In addition to this, declaring types meant that any type-associated errors were either eliminated from the beginning as the developer clearly knew what type would go where or the compiler would rapidly pick them up, allowing the developer to fix them.

"The times I've dealt with JavaScript, it hasn't been good. It's really not clear what types the inputs are and what the outputs are." P10

Some participants indicated that using dynamic languages meant that testing and debugging was harder, with this increasing the longer on a project. One reason given was that without types, it was harder to interpret and understand what was going on inside the code. Using a static language provided better clarity and made it easier to look at other people's code when debugging or doing pair programming.

"Looking through other people's code to see what's happening is a lot more difficult, especially when one person breaks one thing and find out where the break is being caused. It's even worse there's more people working on it. Using a static language might have reduced this." P1

Dynamic – easier to learn. Participants claimed that dynamic languages tended to be easier to learn for those new to programming, learning a new programming language, or who had just joined industry. Participants found that not having to declare variables allowed them to get more work done with less effort which minimized the overhead by having to think less about the semantics of their code.

"I program a lot faster without types. I find them obstructive to my thought process of continuing to design something. It may be because I design things as I go, rather than planning them out." P11

Type declaration was another step that typically slowed things down for these developers. When learning a new language, several participants stated that having to understand and declare types would have slowed down the rate at which they learned. This was because they would have to worry about whether variables were correctly typed in addition to learning and applying new skills and concepts. Conversely, with static languages, the concept of typing and how types

worked for specific languages meant that there was a greater learning curve and thus, take longer to get working code.

"Starting out, it makes it a little simpler. It's a tiny bit of mental labour you don't have to do, meaning you can think at a higher level." P4

New users of dynamic languages felt that they could immediately make progress on their projects and work without having extra consideration of variable types, while new users of static languages believed that there was more of a gap before they could get something working. A language that participants commonly cited as being easy to learn was Python. Even amongst those who advocated for static languages, Python was still regarded as one of the best programming languages to introduce to those who had never done programming before due to the increased complexity typing brings, and relative straightforwardness of the language itself.

3) Structure: Static – enforce more structure within development. Several participants stated that the usage of static languages enforced structure throughout software development. Due to having to declare the types of variables meant that forethought had to be put into envisaging how the code would look before entering it. Some participants believed that having a more structured development process where they had to put forethought into typing and the overall structure of the program meant that there would be less errors and the overall experience would be more streamlined.

"Once we had it up and running and we could show them how everything was organised. In the end, code quality and organisation of code [using Java] was much higher than the JavaScript project we also had running." P7

Using a static language with a more structured development process impacted the experience of developers. For many participants, this is a matter of personal preference. This shapes a developer's experience as it is one thing to have read about structured development and another to apply it, with some developers responding better to more freedom.

Dynamic – provided more flexibility within development. Some of the participants in the study believed that dynamic languages allowed developers to have more flexibility in the development process, without being constrained by type declaration and other enforced structures that arise from static programming languages. Some participants felt that the ability to ignore typing meant that they could spend more time thinking about how to solve the problems presented by the project rather than having to focus on getting the structure and typing right.

"With JavaScript, you can do whatever you want. If you're using Java, you have to adhere to the rules." P1

Dynamic languages provided more flexibility on code structure and are learned by having previous experience with a language, rather than relying on theory. If a developer uses a dynamic language and finds that it allows them to not have to worry about conforming to rules, and they find this approach works for them, it will build a positive experience.

V. DISCUSSION

We now discuss the relationships between each of the categories, with *choice* providing experience, *experience* shaping attitudes, and *attitudes* informing choice.

A. Choices Provides Experience

The relationship between choice and experience is represented by the choices that a developer makes provides them with experience in the future. With a language choice being made and time being spent using the language for either an industry or personal project, the developer builds a greater familiarity with that language. As this familiarity increases, the developer can examine whether the reasons they used to choose that language were in fact justified and met or if usage of that language did not deliver the results they believed it would. Thus, the choice of language provides experience that can be used for future choices. Each of the three factors present within the choice category have an influence on a developer's experience regarding static and dynamic languages.

Project languages often being pre-selected can bring a negative perception of that language to a developer, if they did not enjoy the development process involving it. They may feel they were forced into using that language and that given their own choice, would much prefer to use something else. On the opposite side, this can also build a positive experience. If a developer had tepid feelings about a language, but had to use it and the learning and management structures were there to assist them and they experienced success, this may change their initial negative feelings and convert them into positive ones. The final case is where a developer is ambivalent to the choice of language and their work did not change this. In this instance, there is no positive experience, but no negative experience and they may objectively look at it regarding benefits and drawbacks in the future.

Programming languages chosen based on familiarity will impact experience depending on whose familiarity it was chosen from. If it was the developer's, then this may reinforce a positive experience as they are using a language they are comfortable with. However, if the language was chosen based on a lead developer or manager's familiarity, this results in a similar situation to the previous factor. There can either be a positive influence, a negative influence or ambivalence, dependent on their preconceptions regarding the language and their experience with the development process.

Tooling, IDE, and library support also provides a developer with experience by helping to make programming easier and to simplify complex tasks. They learn what support a language does have and whether it is relevant to the task they were trying to perform. Languages that possess superior support will provide more successful outcomes and developers will look more favourably upon those languages. If their usage of a language involved tapping into its tooling, IDE and library support and this resulted in a positive result, then the experience provided will be positive. If the support was lacking and inhibited a developer's ability to do work, this will result in a negative experience.

B. Experience Shapes Attitudes

The relationship between experience and attitudes is where a developer's previous experience with a programming language then shapes their preconceptions towards that language. Once a developer has used a language and they have experiences with it, these experiences are looked upon when making future choices. They examine their past feelings and sentiments, which feeds into their preconceptions and personal bias. These subjective personal beliefs form the basis of a developer's attitudes to certain programming languages and types of programming languages. Effectively, if a developer has a positive experience with a programming language, then they will have a positive disposition towards consideration and future usage. However, if the experience was negative, then their perception will impact any future considerations. Rather than looking at empirical research and letting previous studies inform them, developer's preconceptions are instead shaped by their experiences. Factors within the experience category all influence a developer's attitudes regarding language usage.

Speed is an indication of how static or dynamic typing contributes to the overall development speed of the project. The compile/build times of static languages slows down larger projects. This reflects experience as it is often not something a developer can foresee, but something that is noticed over the span of the project as it increases in size and scope. With this experience, a developer's attitudes towards static languages is altered. If compile or build times were increased due to use of a static language, a developer may be less inclined to select a static language for another large project. Contrary dynamic languages are good for quick starts and smaller projects. This represents a developer's experience using dynamic languages for small-scale projects. For projects that require working code or deliverables in a short span of time, developers may be more likely to turn towards dynamic languages if they have previous experience of using them for a similar purpose in the past. This then shapes their attitude towards dynamic languages as they view them as being well-suited to small projects or those which require a quick start.

Errors represents the experiences regarding error checking in languages. The first factor that falls within this subcategory is that static languages provide better error checking. This is an indication of a developer's experience with a static language and whether it picks up on errors. It also covers previous experience regarding dynamic languages and their supposed weakness in error checking. This factor can shape a developer's attitude as it provides a clear comparison between the two types of languages. Previous usage of static languages where errors were identified and caught by the compiler and the developer was able to fix them as a result will provide a positive experience. This would shift their attitude of static languages to a more positive slant. Likewise, if previous usage of dynamic languages resulted in less errors being detected and a longer time spent debugging and cleaning up code, then a developer's attitudes towards dynamic languages will be negatively shaped by their error checking experience.

Structure encompasses a developer's experience of how structured or flexible development is using either a static or dynamic language. Static languages enforce more structure within development which indicates how developer's felt static languages affected pre-planning and overall code structure by enforcing type declaration. Structure can have either a positive or a negative effect on a developer's attitude towards static or dynamic languages, depending on their personal preference. If a developer enjoys having rigid development where everything is planned before hand, it will have a positive impact. Otherwise, it will have either no or negative impact. However, structured development was something that more experienced developer's sought. This was usually because they had more experience and acted as project leads and managers. Dynamic languages, however, provide more flexibility within development. This represented whether developer's believed that using dynamic languages would allow them greater flexibility when it came to structuring their code and if it permitted more coding on the fly. Personal preference was significant when it came to whether or not dynamic languages provided a positive or negative impact. Developers that engaged in lots of personal projects enjoyed the flexibility that dynamic languages brought due to less effort required to consider type declaration and time could be put towards getting results. However, this trait was not valued by experienced developers who had acted as project leads, as having a greater degree of pre-planning usually meant that projects were more successful.

It is clear that a developer's previous experience with a static or dynamic language (be it positive or negative) has a significant influence on their attitude towards that type of language in the future. Effectively, the experience shapes their attitudes and moulds their perceptions and preconceptions of static or dynamic languages. This can either be through validating and reinforcing pre-existing beliefs and biases or by changing them and resulting in adopting new languages.

C. Attitudes Informs Choices

The attitudes that developers have regarding certain languages and types of languages are significant in the choice of language. Sometimes the decision to use a certain language or discount it from selection simply boils down to whether a developer likes that language or not. Attitude is difficult to quantify as it deals with a developer's feelings and there are limited concrete ways of measuring this. If a developer's preconceptions of a language are negative, then they will usually not use that language unless there are significant gains to be made from doing so. Likewise, if a developer has a strongly positive perception of a certain programming language, then they will be more inclined to use that language, even if it is not the most appropriate for a project.

Static language partisanship represents a developer's strong positive bias towards the usage of static languages. Participants who advocated for static languages were strongly in favour of them and strongly opposed the usage of dynamic languages. Whereas those who preferred dynamic languages tended to acknowledge the strengths of dynamic languages but accepted

that there were areas where static languages performed better (e.g. error checking). The strong preconception that static language partisanship shows is indicative of how attitudes can inform a developer's choice in what language to use, as those who displayed static language partisanship would be hard-pressed to choose a dynamic language for a project.

Developers with more experience tend to prefer static languages indicates positive bias towards static languages due to their experience. Participants who had less experience in industry tended to prefer to use dynamic languages for a variety of reasons, while participants who had more years of experience tended to opt for static languages. This is another preconception that is held within a more limited group of participants, but can still influence the choice that they make.

Switching between static and dynamic languages was not an issue represents the difficulty a developer may have if two different components of a project are developed using differing languages and their perception of it. For many participants, this was a non-issue as they adjusted rapidly with only a few minor errors being made. However, when making a choice, lead developers may assume that it would be better to have all components of a project use the same language.

A developer's preconceived attitudes towards certain languages or types of languages can impact their choice for a project. If a developer has a negative attitude regarding a programming language, then they are unlikely to select that language even if it is the best suited for a project. The reverse is true for positive perceptions, which may result in choosing a language that is not an optimal fit for a project. Thus, a developer's existing attitudes towards specific languages directly informs the choice of language that they will make.

VI. CONCLUSION

Our aim was to develop an emergent theory of why developers do what they do focusing on the usage of static or dynamic programming languages by interviewing developers ($n=15$) and using Grounded Theory [2], [3]. We produced a *theory of static vs dynamic language choice* that discussed three categories that influenced how developers select languages for projects, the relationships between them, and the factors within these categories. These three categories are: attitudes, choices and experience. Attitudes describes the preconceptions and biases that developers may have in regard to static or dynamic languages, while choice is the thought process a developer undergoes when selecting a programming language, and experience reflects the past experiences that a developer has had with a given language. The relationships between these categories was that attitudes informs choice, choice provides experience, and experience shapes attitudes. This forms a clear link between all three categories and how their factors can shape and influence each other. This is a first cut of the theory and there are several potential future avenues such as interviewing more developers, conducting online surveys, and considering other languages aspects (beyond types systems) to study programming language choice which will further help validate our results.

REFERENCES

- [1] P. Carbonnelle, "PYPL PopularitY of Programming Language," <http://pypl.github.io/PYPL.html>, 2017.
- [2] B. Glaser, *Theoretical sensitivity: Advances in the methodology of grounded theory*. Sociology Pr, 1978.
- [3] J. Holton, "Grounded theory as a general research methodology," *The grounded theory review*, vol. 7, no. 2, pp. 67–93, 2008.
- [4] L. Paulson, "Developers shift to dynamic programming languages," *Computer*, vol. 40, no. 2, 2007.
- [5] L. Prechelt and W. Tichy, "A controlled experiment to assess the benefits of procedure argument type checking," *IEEE Transactions on Software Engineering*, vol. 24, no. 4, pp. 302–312, 1998.
- [6] L. Fischer and S. Hanenberg, "An empirical investigation of the effects of type systems and code completion on API usability using TypeScript and JavaScript in MS Visual Studio," *ACM SIGPLAN Notices*, vol. 51, no. 2, pp. 154–167, 2015.
- [7] S. Okon and S. Hanenberg, "Can we enforce a benefit for dynamically typed languages in comparison to statically typed ones? a controlled experiment," in *ICPC*. IEEE, May 2016, pp. 1–10.
- [8] S. Hanenberg, S. Kleinschmager, R. Robbes, E. Tanter, and A. Stefk, "An empirical study on the impact of static typing on software maintainability," *Empirical Softw. Eng.*, vol. 19, no. 5, pp. 1335–1382, 2014.
- [9] S. Hanenberg, "An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time," *ACM SIGPLAN Notices*, vol. 45, no. 10, pp. 22–35, 2010.
- [10] A. Pano, D. Graziotin, and P. Abrahamsson, "What leads developers towards the choice of a JavaScript framework?" *arXiv preprint arXiv:1605.04303*, 2016.
- [11] L. Meyerovich and A. Rabkin, "Empirical analysis of programming language adoption," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 1–18, 2013.
- [12] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in GitHub," in *FSE*. ACM, 2014, pp. 155–165.
- [13] A. Martin, R. Biddle, and J. Noble, "XP customer practices: A grounded theory," in *Agile*, 2009, pp. 33–40.
- [14] ———, "The XP customer team: A grounded theory," in *Agile*, 2009, pp. 57–64.
- [15] S. Adolph, W. Hall, and P. Kruchten, "Using grounded theory to study the experience of software development," *Empirical Softw. Eng.*, vol. 16, no. 4, pp. 487–513, 2011.
- [16] R. Hoda, J. Noble, and S. Marshall, "Grounded theory for geeks," in *PLOP*. ACM, 2011, p. 24.
- [17] ———, "Developing a grounded theory to explain the practices of self-organizing agile teams," *Empirical Software Engineering*, vol. 17, no. 6, pp. 609–639, 2012.
- [18] S. Dorairaj, J. Noble, and P. Malik, "Understanding lack of trust in distributed agile teams: A grounded theory study," in *EASE*, 2012, pp. 81–90.
- [19] M. Waterman, J. Noble, and G. Allan, "How much up-front?: A grounded theory of agile architecture," in *ICSE*. IEEE, 2015, pp. 347–357.
- [20] R. Hoda and J. Noble, "Becoming agile: A grounded theory of agile transitions in practice," in *ICSE*. IEEE, 2017, pp. 141–151.
- [21] P. Montgomery and P. Bailey, "Field notes and theoretical memos in grounded theory," *Western Journal of Nursing Research*, vol. 29, no. 1, pp. 65–79, 2007.
- [22] A. Pang, "Why do programmers do what they do," 2017, Honours Report. Victoria University of Wellington, New Zealand.

API Designers in the Field: Design Practices and Challenges for Creating Usable APIs

Lauren Murphy
University of Michigan
laumurph@umich.edu

Mary Beth Kery
HCII, CMU
mkery@cs.cmu.edu

Oluwatosin Alliyu
Haverford College
alliyut@gmail.com

Andrew Macvean
Google, Inc.
Seattle, WA
amacvean@google.com

Brad A. Myers
HCII, CMU
bam@cs.cmu.edu

ABSTRACT—Application Programming Interfaces (APIs) are a rapidly growing industry and the usability of the APIs is crucial to programmer productivity. Although prior research has shown that APIs commonly suffer from significant usability problems, little attention has been given to studying how APIs are designed and created in the first place. We interviewed 24 professionals involved with API design from 7 major companies to identify their training and design processes. Interviewees had insights into many different aspects of designing for API usability and areas of significant struggle. For example, they learned to do API design on the job, and had little training for it in school. During the design phase they found it challenging to discern which potential use cases of the API users will value most. After an API is released, designers lack tools to gather aggregate feedback from this data even as developers openly discuss the API online.

Keywords—*API Usability, Empirical Studies of Programmers, Developer Experience (DevX, DX), Web Services.*

I. INTRODUCTION

An Application Programming Interface (API) describes the interface that a programmer works with in order to communicate with a library, software development kit (SDK), framework, web service, middleware, or any other piece of software [1]. Given their ubiquity, APIs are tremendously important to software development. With the rise of web services, APIs are projected to rapidly grow into a several hundred billion dollar industry [2] [3]. APIs are increasingly a primary way that businesses deliver data and services to their user-facing software and also to client software at other companies that purchase and receive that business's services via APIs [2] [4]. As of April, 2018, programmableweb.com listed over 19,500 APIs for Web services, with hundreds more being added each year.

The study of API Usability [5] has often focused on the programmers who use APIs in their own code, known as *API Users*. The developer experience (DX or DevX) is significantly impacted by the quality of the APIs they must use. However, the usability that these programmers experience with an API ultimately stems from how well the API and its resources were designed in the first place. Very little attention so far has been given to the API design processes or to the *API designers* who are making these design decisions. With an increasing number of companies creating APIs, it is important to have an understanding of API design in the field. What is the current process of designing and producing APIs in real

organizations? What current roles do human-factors and usability play in design decisions? What are the open challenges that API designers face?

In order to better understand the real-world design process of APIs and challenges of API designers, we conducted interviews with 24 professional software engineers and managers experienced in API design across 7 major companies. We found interesting insights about how APIs are designed and ways to improve this process. For example, we found that API design is learned most often through practice and can be cultivated through exposure to API design reviews. These reviews preferably occur at multiple stages in API development. Early feedback from users on an API's design and future use cases will result in a better design, but was reported to be challenging to obtain. User testing - even quick, informal studies - can be greatly beneficial. Documentation that is not discoverable can turn away potential customers. More customization is needed for general-purpose documentation and SDK generators but success has been found with custom ones. Other findings are discussed throughout.

II. RELATED WORK

A. API Usability

Since APIs are a form of user interface, considering the usability of that interface is important [6]. API usability has been shown to impact the productivity of the programmer, the adoption of the API, and the quality of the code being written [1]. If used incorrectly, research shows that the resultant code is likely to contain bugs and security problems [7].

However, APIs are often hard for programmers to learn and use [1], with prior work identifying many causes, including the semantic design of the API, the level of abstraction, the quality of the documentation, error handling, and unclear preconditions and dependencies [8]–[13]. Additionally, changing an API after it has been deployed is difficult, due to its potential to break the software that depends on it [14]. All of this combines to make the API design process critically important.

B. API Design

There are a number of decisions API designers must make to create an API [15], and quality attributes that must be

evaluated [1]. The impact of some API design decisions have been explored, providing API designers with empirically based guidelines to follow. For example, the use of the factory pattern [16] and required constructor parameters [17] were shown to be detrimental, and method placement was shown to be crucial [18]. Additionally, recent work has looked at defining metrics for encapsulating and measuring API usability, allowing for larger scale quantitative assessment of an APIs design [19], [20]. In some cases, as much as 25% of the variance in the proportion of erroneous calls made to an API could be predicted by modeling just 7 structural factors of the API's design, including the number of required parameters in the API call, and the overall size of the API [21]. On this vein, several recent research projects are exploring new forms of static analysis tools to detect API usability issues [41].

To collect and standardize common API design decision, a number of major companies like Google [22] and Microsoft [23] have released API Style Guides. A recent review of these guides identified both core principles, and inconsistencies contained their advice to designers [24]. Finally, although online API style guides are the most up-to-date resource for API designers, a number of useful principles for API design can be found in software engineering books [25][26], as well as classic theories for evaluating usability, such as the Cognitive Dimensions of Notation [27].

C. Understanding The API Design Process

There are a number of examples of the positive impact that can result from using traditional HCI methods during the API design process, including usability studies, design reviews, and heuristic evaluations to aid in understanding and improving the APIs' usability [27]–[31]. However, more broadly understanding the needs of the API designers, and the process they go through while they design, implement, release, and maintain an API, is less well explored. Macvean et al. discuss part of the web API design process at Google, which includes an expert review of a proposed API design. The goal is to ensure consistency and quality while providing API designers with the ability to consult with API design experts [28]. Henning stresses the importance of education for successful API design, arguing that good API design can be taught [32].

While there has been much work done in understanding the software engineering process in general, e.g., the role of knowledge sharing within organizations [33], the impact of distance between engineers [34], and the importance of well defined tasks [35], understanding the specifics of the API design process, and the barriers facing API designers, have not been previously studied.

III. METHODOLOGY

To better understand the processes, challenges, and constraints of API designers, we conducted structured interviews with practicing API designers from industry. First,

we iteratively constructed 36 questions through discussion with several of our current and former collaborators who have API research or qualitative research experience. The resulting questions (many of which had follow-up subparts) focussed on a broad range of API design topics, including what a good API is, the design processes they follow, difficulties faced when designing, and how to improve this process in the future. (The full questionnaire is available in the supplemental material and at natprog.org/papers/InterviewScript.pdf.) We interviewed 24 different API designers from seven large companies, from various positions within the companies. Each interview lasted from 60 to 120 minutes, and was audio recorded which was complemented with detailed notes. The interviews were all performed via video conferencing, since the interviewees were remote (with participants from USA, Asia, and Europe). All the interviews took place during working hours, and there was no compensation offered.

We transcribed a total of 38 audio hours of interview data for our analysis.. For two interviews where the audio failed or was too poor quality to transcribe, we relied on the detailed notes for analysis. Following grounded theory methodology [36], the 1st author first sampled four interviews to do open-coding which generated 505 codes. Those codes were reduced down to 41 specific labels, such as Usability Factors, Company Processes for API Lifecycle, and Audience & Use Cases. At first, we followed standard inter-rater reliability [37] as we expected the priorities of the designers to be fairly consistent. However, we found a rich diversity from the designers, and thus, the use of inter-rater reliability proved to be a misstep. Initially, the 1st and 2nd author each independently coded a new sample of five analyses (20% of the data), receiving a low Cohen's Kappa of 0.55 [37]. Both authors discussed disagreements, refined the code book, and repeated the process on a new sample of five interviews. With a moderate Cohen's Kappa of 0.71 [37], the two authors labeled all remaining interviews together, allowing for multiple labels where needed, as decided through discussion and consensus. Afterwards, our analysis followed 'data-driven' thematic analysis [42] where we clustered our coded data into themes.

IV. PARTICIPANTS

We used snowball sampling to recruit: first inviting personal contacts to participate, and they in turn asked appropriate people at their companies who were involved in API Design. Between June 2017 and February of 2018 we interviewed 24 participants from seven large companies: one financial company and six tech companies. Participants had an average of 16 years programming experience, with an average of 9 years API design experience. Five participants are currently in upper management, while all others were software engineers. Participants will be called P1 to P24. Quotes below are filtered to protect anonymity, and instances where a participant mentions company-specific terms will be replaced by "X". Table 1 summarizes the types of APIs that participants worked

on. Note that 9 of 24 worked on more than one type. All had designed APIs that were in use by many programmers, ranging from hundreds to millions of users.

TABLE 1. TYPES OF APIs THAT PARTICIPANTS WORKED ON

REST API	P1, P2, P3, P4, P5, P6, P7, P15, P17, P18, P19, P20, P21, P22, P23, P24
Other Web API	P1, P5, P6, P7, P8, P16, P17, P18, P21, P23
Other Library or SDK API	P6, P9, P10, P11, P12, P13, P14, P19

TABLE 2. AUDIENCE OF THE APIs

Public API	P1, P2, P3, P4, P6, P7, P8, P9, P11, P12, P14, P17, P19, P20, P21, P22, P24
Internal API	P1, P5, P6, P10, P12, P13, P15, P18, P20, P23, P24
Gated API	P15, P16, P17, P22

In Table 2, Public APIs are those available to any developer to use. Internal APIs are built in-house and used only within a company. Gated APIs have a select set of enterprise customers who directly communicated their feedback to the API team.

V. RESULTS

A. Learning to Design APIs

All API designers we talked to had primarily learned to design APIs through work experience. Raw experience with designing APIs and learning from colleagues was prized over any form of formal educational resource. The designers' education level ranged from a PhD to a high-school diploma. Some had taken software engineering courses, yet only four participants had learned anything about API Design in school.

Even when interviewees came from large software companies, they reported that API design was a specialization of a relatively small group of people in their company. API design experts are relatively rare. We asked participants what would differentiate a good API designer from a regular software designer. API designers must have a strong understanding of software engineering, but *also* have the ability and personal drive to stay focused on their user's perspective as their primary goal:

"A good API designer would put [themselves] in the shoes of [another] person who is actually going to use the API whereas a good software designer would basically look at it from [their own] perspective if the design is good or whether the design is scalable." - P21

As a solution to foster more experts, five participants reported that their company had a mentorship program, in which novice API designers (typically any engineer interested in the topic) could sit in on the expert API design reviews.

Implications: API design is recognized as a difficult and specialized skill with few training resources, so more training material is needed. On the job, the experience that expert designers need could be scaffolded much like many other

design disciplines are taught, with creation and critique exercises. Novice API designers would practice creating APIs (plausible design exercises that are not on the critical path of a real product) and have their work critiqued in API design reviews. Additionally, since a good API designer needs an intuition for user experience, this may be trained by giving interested developers exposure to basic user experience testing and usability methods that have been well established in UX.

B. Using Existing Guidelines

Today, API guidelines published online by various organizations are the primary authoritative source for design standards and practices [24]. Many companies look to and copy from API guidelines of industry leaders, consistent with observations from Murphy et al. [24]. Nine participants we spoke with were active contributors to API guidelines at their organization. Although they might use the principles from the API guidelines, not all designers we spoke to had actually read the entirety of their company's guidelines, instead using it as a general-purpose reference material to look up specific design questions as they arose.

However, some participants found that the broad nature of the guidelines left them still needing guidance when it came to specific decisions. Participants disagreed about whether the guidelines they had were sufficient to really serve as a design tool or if and how they should be improved. This disagreement often centered around the inclusion of recommendations specific to use cases. Five participants reported making custom guidelines for their team's use cases to supplement the company-wide guidelines.

Some companies enforced their API guidelines through the use of code reviews and "linter" tools that check for specific guideline requirements. The purpose of this enforcement is to ensure a base level of API usability across the company, and also ensure API users have a *consistent* experience if they use multiple APIs across the company to avoid each API having its own learning curve. When a team or company did not systematically enforce the guidelines through a built-in part of code review culture or linter tools, adherence was difficult:

"When we got acquired by [Company X], and we found out about this [Company X] standard, I was actually relieved... It was very hard [before that] to get our service engineers to consistently represent certain concepts exactly the same way. And the [Company X] REST standard just laid that out." - P24

Finally, as discussed further below, designers often have multiple competing design concerns and constraints to balance during API design, making following *all* guidelines much harder than it might appear. Specifically, new designers struggle with knowing the relative importance of different guidelines, and must learn through mistakes when to adhere or deviate from a rule. One participant suggested that examples and case studies of specific guidelines in the API would help

them learn how and when to apply them. Though no other participants mentioned case studies specifically, four others wished for better rule-by-rule rationale to be included.

Implications: Best practices include following API design guidelines, which might be locally defined or adapted from other companies. At their core, design guidelines are collected wisdom from many different developers over time, so that repeated design decisions do not need lengthy discussion every time, and so that informed decisions can be made about when to break from convention. Company-wide and industry-wide API guidelines are not a one-size-fits-all rulebook, and where needed developers must supplement with team-specific or product-specific guidelines. By documenting their problem-specific API design decisions, a team can help achieve *consistency* in their future design decisions, which is a core tenant of how to make APIs (and any other form of user-interface) more easily learnable by users.

C. Who designs an API?

When asked what roles in their organization are involved in API design, designers said new APIs are typically requested from upper management or solution architects and first specified by project managers. More rarely, an engineer or product manager could propose an API be created, and some companies had pathways where that person could write up a proposal for the higher management to approve.

“So there’s a theoretical view that says the offering managers or the product managers are the ones who are deciding what the functionality should be, and the API designers and the engineering team in general are just deciding how to convey that functionality how to present it. But in reality the two are much more closely constrained.” - P2

When an API’s domain of use is not a highly technical engineering domain, such as providing product data or financial data, key design decisions come first from a product manager. Highly technical engineering APIs, such as those whose target users are database or network engineers, are more heavily designed by engineers. Although three participants mentioned trying to involve user experience (UX) experts or technical writers in the process to help choose good abstractions and naming, the technical knowledge needed to design for code developers is a major barrier:

“The big problem with bringing UX people in is that they don’t have a background in APIs or [even], in some cases, in programming. ...They get overwhelmed by either the people or what it is that we’re putting in front of them” - P1

Thus when it comes to usability concerns around the developer experience of what specific code the API users will need to write, the brunt of responsibility falls on the company’s software engineers to seek understanding of and design for their users.

Implications: Whomever designs the API is responsible for figuring out their API users’ needs and keeping those needs central to the design process. Ideally, API design should be done by an interdisciplinary team, however the challenge remains of teaching expert software engineers enough user-centered design and teaching expert UX designers enough software engineering that these two disciplines can work effectively together.

D. Designing an API from scratch

No participants complained of insufficient engineering expertise in their teams to specify the functions and datatypes for an API. One participant even said that they were happy to give API design tasks to a junior engineer as a learning opportunity. Rather, a major challenge for three participants was that teams commonly poured far too much time into designing and specifying *for the wrong use cases*:

“We often spend lots of time worrying about these edge cases that in essence zero or nearly zero people end up using” - P2

The business value of the API which is expressed at a high level like to “provide email data” or “provide access to cloud computing” or “provide a language specialized for X” is generally too abstract to anticipate the specific use cases and constraints the customer developer is going to have for an API. At the initial stage, designers aimed to release the API as quickly as possible, with a minimum amount of functionality needed to get the API product on the market. Participants said poor quality “bottom-up” API design occurs when, lacking real use case data, the engineering team designs around what is most straightforward to implement, which means that the API design mirrors much more the underlying *implementation* of the API than how customers want to use it.

“Knowing how many people are using your API and for what, is... often difficult to do” - P14

For internal APIs, where all users of the API are in house, the risk of getting an API wrong the first time is fairly low:

“I start with an API that’s just does the minimum possible and if it doesn’t work we can change it later.” - P9

In contrast, for publicly released APIs, *all* decisions good or bad quickly become canon in users’ code, since API users have been known to depend on aspects of the API as low level as the line numbers reported in error messages:

“If you change anything you basically break people ... so you need to plan much more, how should it be used, will it be used and you cannot change it afterwards so it’s much much harder.” - P9

It should be noted that while many types of software have to deal with updates and backwards compatibility, the case with publicly released APIs is quite severe. The use of an API is baked into the API users’ code, meaning that any change to the API has the potential to break the customer’s code and

require many engineering hours from the customers to update their code with the new API version. To avoid “breaking changes”, it is important that the API designers get core abstractions and core methods of the API correct the first time. As in any usability decision, developers must prioritize making some use cases easier than others. In an API, the core abstractions should ideally fit the real-life core use cases as closely as possible, because this permanently affects the current and future usability of the entire API unless serious breaking changes are possible:

“Over-specifying things can sometimes be troublesome. For example, we’ve had the concept of X in our APIs, and it over-specified the X and had things in there that aren’t used. ... You can’t take them away because that breaks existing clients.” - P23

Implications: Before getting too far into construction details of the API (the things that are covered in API design guidelines) like naming, pagination, etc., it is critical to check your team’s understanding of the API’s real life client use cases, since it will often be difficult or impossible to change these later. This can be achieved by getting users involved early on in the design process and continually obtaining feedback from them throughout this process.

E. Getting User Feedback on Initial API Design

Eighteen designers reported that they start developing their products with common use cases in mind. In the case of gated APIs developed for specific customers, designers had the ability to directly communicate with their users to understand what the use cases would be. When the APIs were meant for internal use, getting feedback about use cases was also direct:

“Most of my work have been more internal... often time we’re coming in with a very better understanding of the users, we can just talk with them directly.” - P5

For public APIs, (incidentally where the risk of getting an API wrong is also highest) participants most often reported that they tried to imagine themselves as future users and then built use cases largely on intuition:

“For cases where the API is new and there is essentially no usage, then obviously at that point you’re relying on either your own experience as a designer or what use cases you can manage to glean from people that say, ‘yes, I’d like to have a thing like that,’” - P14

Two participants mentioned creating “user stories” to base their design around. One such designer compared these user stories to personas, which are often a component in UX design. One designer said they used cognitive dimensions [27] as a method for API design to think through a user’s experience. Another designer, who happened to have some training in human-computer interaction, took design ideas to informally test with any other developers in the lunchroom:

“I was working on a design for [API X] just the other day and I was running really quick and dirty user studies in the café during lunch and it helped! I got tons of questions answered. I’ve gotten other people to do it, and ... it has affected the API design. Before they put tons of time in crafting a metric name for some monitoring API, right? Like taking their candidate names in front of people and having them explain what those metrics are.” - P1

Even though P1 worked on a public API, getting feedback from a broad range of developers inside the company (but outside of the API X team itself) gave P1 more insight about the use cases and perspectives. More generally, in order to test the ease of use of an API design, participants mentioned that it is a good sign if a developer (an API user, an API reviewer or just a convenience sample of developers in the company) is able to read through the API specification and have a good understanding of what the API does based on the names alone without relying on documentation. API Peer Reviews [31], which have someone interpret the API by names alone is also a usability test that could easily be performed early in the API design phase where only the specifications, and not the concrete implementation, exist.

Gathering use case data at this early sketching phase of designing an API was challenging to all participants. However, once the API was an implemented prototype, designers were comfortable using beta-testing and obtaining feedback that is typical of most any new software:

“We had a lot of clients with different needs so the first thing we did was we built out a very lightweight prototype of it where we just packed it together and kind of you know put something out and send it out ... as soon as we could to a bunch of different teams with various degrees of expertise and various use cases” - P5

Five participants also did formal usability testing where they observed and measured developers trying out the API:

“We get with our group of developers that build an app and they start working on their APIs, and we measure how long it takes for them to get from 0 to 200... we use that 0 to 200... at different stages. One is during the development side. Two is ... before going into production. And during production” - P22

Here “0 to 200” refers to how long it takes a developer to get a 200 value returned from the API’s web server, signaling that the call was processed without error. This metric, also known as Time to Hello World [9], was used by two designers as a measure for how easy it is to use an API. Four designers had done usability studies in the past but found performing studies to be too time and resource expensive to do regularly.

Implications: Best practice requires testing an API with users early, even when the API is not yet implemented or even fully designed. Beta-testing implemented prototype APIs is an existing software engineering practice and should be done. Formal usability testing is rare and also time consuming, but

relying on simple measures like time from 0 to 200 may make usability studies less daunting to perform. In the face of low resources or time, quick feedback from peers *outside* of the API team is a great resource. Teams might try informal user testing like P1's lunchroom exercise, or a hackathon style lunch where developers from inside the company come for food and sit down with members of the API team and follow a “think aloud” protocol [43] where each “user” developer walks the team member developer how they would use the API in its current design. The flexibility of this is that the API can be fully implemented, or just a list of methods on a napkin, and all that matters is observing how users attempting a real task approach your API. Even in these informal settings, however, it is crucial to follow core tenants of formal usability testing so as not to ruin the validity of the exercise, for instance: 1) Predefined tasks for the user to do with your API so that you can later compare how different users respond to the same circumstances. 2) Avoid correcting or overly teaching the user how to use your API when they try it out (even if they mess up) since your API must stand on its own and your real users will not have you sitting behind them. 3) Be open to negative feedback, even if you feel the user doing the task is not knowledgeable or “smart” enough to understand your design (your final API users may very well be the same).

F. API Design Review: Key to Ensuring Quality

All participants reported using design or code reviews as part of their API development process. To help focus the reviews, twenty participants mentioned using automatic tools, like FXCop [39] or Clang-Tidy [40] and custom linters, which identify low-level issues that then do not need to be covered in the review. Instead, they can focus on broader, more subjective issues such as intent, customer workflows, usable naming decisions, and how the code fits consistently with the rest of the company’s codebase:

“A manual review should focus on usability and like intangibles about it - about use cases and APIs and then automated tooling should focus on the annoying stuff, like did you name this correctly or you used casing inconsistently, or this is paginated and this isn’t paginated” - P18

For most of the companies we talked to, there was a group of API design experts who performed the design reviews, especially for public APIs. Some companies had a small group while others had a much larger group who handled the task. One participant mentioned that their company had a single person act as the expert reviewer for their division, to maintain higher levels of consistency.

Another participant recounted how, before even looking at the API, they would try to understand the problem domain and the resources and relationships that exist within it. After that is understood, the reviewer would check to see how well the API design captured those relationships. However, others noted that reviews failed to serve their purpose when too much high-level design discussion of the API meant they never sat

down to concrete code examples that the API’s user will have to work with.

The point at which design reviews are introduced into the overall API development process varied widely among participants. For some, the reviews were incorporated as early as when they were sketching out the core abstractions and naming. Five participants strongly encouraged this early review feedback. Others held a different kind of final review of the product with a committee at the end of development.

Implications: Best practice requires having design reviews preferably at multiple stages of the API design. These reviews should be performed including API design experts. There are multiple kinds of review. For instance, in a high level conceptual review, reviewers should consider whether the structure of the API makes sense and accurately reflects the relationships of the problem domain. In a code experience review, the reviewer should try to write or read actual code snippets for a real use case, to review the quality of the source code that must be generated to achieve a user’s task.

G. Web and REST APIs

Though a diverse group of designers were interviewed, the majority of participants were involved in web and REST APIs. Some of the usability concerns that were identified were specific to those kinds of APIs. For instance, pagination presented a specific concern for participants in terms of how closely the API’s representation of the data should match a consumer-facing UI’s representation of the data:

“The web API might return 10 results per page just like the UI, but if when, then, should the client library do that? Or should the client library return something that looks like an arraylist in Java, call next, and you just get the next one? I haven’t seen that actually well handled.” - P1

Another designer brought up the difficulty in representing data when multiple lists are involved in a response:

“Pagination is really good when the response has one single list, that needs to be paginated but if it has multiple lists then it becomes more difficult from the service perspective as well as from a customer perspective, to understand where the boundaries are between the two lists and if these two lists are related, that becomes really difficult as well.” – P21

Designers also had usability concerns regarding the fields contained in the response sent back to users. Determining what information is necessary to cover a range of use cases and what is excessive is a challenge that designers face. Providing too much information could overload the response object sent back to users, but providing too little information means some use cases will not be met. One participant tried filtering to allow users more control regarding what the response contains. Though they mentioned having set patterns for filtering, they also wondered what potentially better ways to structure filters would be. A poorly structured filter could

cause backwards compatibility issues as an API grows in complexity and new users require different information.

Implications: Web and Rest APIs are an enormous area of active API development, but there are still some gaps where there are no obvious design best practices covering how to best chunk and filter data to return to the user. Responses should be designed to help users understand boundaries between relationships in the data that they are requesting.

H. Documentation & User Starting Experience with an API

The initial experience with an API was a major usability concern with seven designers, because from a business perspective, a developer's first encounter with an API largely determines if they will adopt it. If first time users face too many learning barriers, then businesses miss out on customers who attempted to use their products:

"Getting started is something that I have seen as a big challenge for developers starting to use [API X] because there are lots of concepts involved when it comes to cloud ... a lot of these terminologies, cloud-based terminologies and service-specific terminologies. So that is where I have seen the biggest problem or challenge." - P21

API designers often felt that documentation suffers from discoverability issues. This issue may arise because the names used in the API are more abstract than the use case that a user has in mind - for example a user looking to draw a circle may search documentation for "circle" but the correct query would be 'shape' - or it could occur due to a lack of conceptual knowledge in a domain [44]:

"The [name] we came up with is kind of an analogy... But you just know there are people out there who start typing and hit autocomplete and cross their fingers and it doesn't come up and they just write it themselves. So I think that's always a tough thing is how do we make these things findable." - P11

Examples of high quality API documentation however were brought up when participants discussed APIs they admired, such as the Stripe API which has three panels containing a list of methods, details about those methods, and code snippets that demonstrated common use cases. Not all documentation lives up to the ideal though. Designers admitted that they rely on Stack Overflow posts or community-built tutorials to fill in for gaps in their documentation, and so supplemental material may complement company-provided documentation.

"I do think developers and API designers treat documentation as an afterthought." - P8

Implications: Documentation and support resources are crucial to onboarding a user to the API, and research is needed on optimal ways to display documentation. With new APIs that have the "cold start" problem of no existing support on online communities like Stack Overflow, designers should create example projects and code snippets to demonstrate the

API so that new users see how the conceptual pieces of the API come together in concrete use cases.

I. Feedback & Usage are Hard to Measure, Hard to Interpret

Once an API is released, designers and their team highly value feedback to improve the usability and quality of the API:

"I would like to know where they are being slowed down or points that are particularly frustrating and what parts take a long time to figure out or find." - P3

Feedback about APIs can be surprisingly hard to gather and interpret. The best case was with internal APIs, where the team had good access to talk to their users directly and with the added benefit of being able to look at their users' code:

"One nice thing about working at [Company X] is that... you can actually just look and see 'okay these are all the places inside [Company X] that this API is being called and how it's being called.' You can look at the code, you can get a count of how many places there are and so forth and that has been incredibly useful; it means that my decisions, my thoughts on how things ought to be organized are considerably more informed than they would be otherwise." - P14

Designers of public APIs have little direct access to their users and typically had a far larger user base. An exception is large enterprise users of a public API, who more often have a direct form of communication with the team. For web APIs, server-side metrics offered counts about which API were most often used, but only vague clues about use cases or usability:

"Sometimes a high amount of error codes means that callers do not understand ... or sometimes they just don't care and they'd rather just get the error conditions back to check with their call. So it can be difficult to parse out their intents." - P5

Furthermore, designers said that counts of "how often is this API method called" were often unhelpful to infer real use cases, since it is unknown what the programmer does with the data once they receive it from the API.

For public APIs, there are often ample examples of usage on Github and many programmer questions reported on Stack Overflow, but it can be difficult to identify what is useful. Despite the large amount of data from online programmer communities, six designers we talked to had both actively monitored and failed to glean useful insights from Stack Overflow and similar places. A major challenge is that there are no tools available for API teams to consume community data in aggregate. One participant had developed a way to mine instances of the API usage from Github repositories, but it should be noted that a custom mining program is not trivial, and still leaves open the problem of aggregating examples to yield insight into where misuse or usability problems may be.

The most reliable source of public feedback was reported to be GitHub issues on public API repositories. Although

designers reported still needing to sift through considerable noise to gleam usable design information, users often post feature requests on GitHub issues which gives designers valuable information and suggestions about future directions for the API. Although there is often an abundance of feature requests, the problem is not so much aggregation. Instead, feature requests lead to healthy debates in the API team about what the API's scope and core design principles should be:

"You know, what's the best API we can design that would serve a reasonable number of people, what would the code look like then? And then how many times does this come up? So you know there are limitless number of feature requests that people ask for, even reasonable things we could think of and we end up having to not add at all for various reasons." - P11

Fourteen designers mentioned other feedback mechanisms like chat channels or customer surveys, but the latter often had too low response sample and too high self-selection issues to yield information about the user population in aggregate.

Implications: Designers of public APIs struggle to get usage feedback after their API is released. Although there are large sources of online programmer community data using a given API, designers currently need better tools to help gather and interpret that data in aggregate. Best practice seems to be to collect anecdotal feedback through Stack Overflow, Github issues, surveys, and direct contact with customers.

J. Automatic Generation of SDKs & Documentation

A web API, on its own, is expressed as textual messages sent to a server. So to improve the usability of web APIs, companies often build Software Development Kits (SDKs) which provide a library wrapper in a certain language for using that API. Interviewees' companies built SDKs for one to as many as nine different languages for a single API. To scale to supporting many languages, some companies use tools that, given a formal specification of the API, will generate the SDK in the various target languages (some also generate documentation). However, generated SDKs were a contentious topic. Interviewees in favor of this approach said that the generated libraries then have consistent naming and abstractions across the SDKs by avoiding idiosyncrasies of individual developers. Interviewees from two companies reported great success with code generation:

"We did a little bit of investigation into generated SDKs early on and thought they were complete garbage and walked away for two years. The generation technology that we're using today... we wrote our own generator that will generate APIs that are indistinguishable from the hand written APIs." - P2

Five participants used Swagger API tooling to document an API's design or preview what the API *might* look like in a certain language, but no participant reported using Swagger's code generation tool for their SDKs. Interviewees who avoided generated SDK complained of low quality of

generated SDKs or inflexibility for the generator to meet their company's specific requirements (like security).

The two companies that routinely and successfully generated SDKs had: 1) internal custom-made generators that were specific enough to match the company's security policies, style guides, etc., and 2) access to lots of processing power. One participant reported the sheer processing time needed to generate these SDKs limited the number of refinements the team could make.

Since individual languages have vastly different styles and idioms, some participants raised concerns about language specific usability of the generated code. While not impossible, good multi-language usability requires significant work on the generator per language:

"It's possible to generate SDKs in multiple programming languages from a single model and make them feel idiomatic for each one of those languages you generate for. But what you have to have are experts at each one of those programming languages. We actually have dedicated teams for each language that maintain the code-generator." - P18

Implications: Technologies to automatically generate SDKs and documentation have greatly improved over the last few years. However, to achieve good usability for different languages requires generation engines carefully tuned by experts in those languages. General-purpose generators like Swagger need to be far more tunable to match the success of bespoke in-house generators, to give designers the freedom to match their company requirements and usability concerns.

VI. LIMITATIONS

This study was limited by the number of participants and companies that they represent, thus may not generalize to all designers or companies. All participants self-selected whether to participate, so participants are primarily designers who have a strong interest in API design quality and API usability.

VII. CONCLUSIONS

Even though there exists some literature, papers, and blogs that talk about API design processes, tools, and guidelines, the interviews that we conducted provide insights into the real world situations and needs. We hope that companies, researchers, and veteran and new API designers can use the information in this paper to improve their own processes, create well-designed APIs, and create new tools and guidelines to help in the design process for future APIs.

ACKNOWLEDGMENTS

This research was supported in part by a grant from Google, and in part by NSF grant CCF-1560137. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the funders.

REFERENCES

- [1] B. A. Myers and J. Stylos, "Improving API usability," *Commun. ACM*, vol. 59, no. 6, pp. 62–69, May 2016.
- [2] Keerthi Iyengar, Somesh Khanna, Srinivas Ramadath, Daniel Stephens, "What it really takes to capture the value of APIs," McKinsey & Company, Sep. 2017.
- [3] Press Release From Research, "\$200+ Billion Application Programming Interfaces (API) Markets 2017-2022: Focus on Telecoms and Internet of Things," 07-Sep-2017.
- [4] Bala Iyer, Mohan Subramaniam, "The Strategic Value of APIs," Harvard Business Review, Jan. 2015.
- [5] B. A. Myers and J. Stylos, "Improving API usability," *Commun. ACM*, vol. 59, no. 6, pp. 62–69, 2016.
- [6] B. A. Myers, A. J. Ko, T. D. LaToza, and Y. Yoon, "Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools," *Computer*, vol. 49, no. 7, pp. 44–52, Jul. 2016.
- [7] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith, "Rethinking SSL development in an appified world," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 49–60.
- [8] C. Scaffidi, "Why are APIs difficult to learn and use?," *Crossroads*, vol. 12, no. 4, pp. 4–4, Aug. 2006.
- [9] A. Macvean, L. Church, J. Daughtry, and C. Citro, "API Usability at Scale," in *27th Annual Workshop of the Psychology of Programming Interest Group-PPIG 2016*, 2016, pp. 177–187.
- [10] M. Robillard and R. DeLine, "A field study of API learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.
- [11] D. Hou and L. Li, "Obstacles in Using Frameworks and APIs: An Exploratory Study of Programmers' Newsgroup Discussions," in *2011 IEEE 19th International Conference on Program Comprehension*, 2011, pp. 91–100.
- [12] M. Piccioni, C. A. Furia, and B. Meyer, "An Empirical Study of API Usability," in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, 2013, pp. 5–14.
- [13] M. F. Zibran, "What Makes APIs Difficult to Use?," *International Journal of Computer Science and Network Security*, vol. 8, no. 4, pp. 255–261, 2008.
- [14] A. Macvean, M. Maly, and J. Daughtry, "API Design Reviews at Scale," in *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems (CHI EA '16)*, 2016, pp. 849–858.
- [15] J. Stylos and B. Myers, *Mapping the Space of API Design Decisions*. 2007.
- [16] B. Ellis, J. Stylos, and B. Myers, *The Factory Pattern in API Design: A Usability Evaluation*. 2007.
- [17] J. Stylos and S. Clarke, *Usability Implications of Requiring Parameters in Objects' Constructors*. 2007.
- [18] J. Stylos and B. A. Myers., "The Implications of Method Placement on API Learnability," in *Sixteenth ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE 2008)*, 2008, pp. 105–112.
- [19] T. Scheller and E. Kuhn, "Automated measurement of API usability: The API Concepts Framework," *Information and Software Technology*, vol. 61, pp. 145–162, 2015.
- [20] G. M. Rama and A. Kak, "Some structural measures of API usability: SOME STRUCTURAL MEASURES OF API USABILITY," *Softw. Pract. Exp.*, vol. 45, no. 1, pp. 75–110, Jan. 2015.
- [21] A. Macvean, L. Church, J. Daughtry, and C. Citro, "API Usability at Scale," in *27th Annual Workshop of the Psychology of Programming Interest Group - PPIG 2016*, 2016, pp. 177–187.
- [22] Google, "API Design Guide," 21-Feb-2017. [Online]. Available: <https://cloud.google.com/apis/design/>. [Accessed: 2017].
- [23] Microsoft, "API design," 13-Jul-2016. [Online]. Available: <https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design>. [Accessed: 2017].
- [24] L. Murphy, T. Alliyu, M. B. Kery, A. Macvean, B. A. Myers, "Preliminary Analysis of REST API Style Guidelines," in *8th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU'2017) at SPLASH 2017*, p. to appear.
- [25] K. Cwalina and B. Abrams, *Framework Design Guidelines, Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Upper-Saddle River, NJ: Addison-Wesley, 2006.
- [26] J. Bloch, *Effective Java Programming Language Guide*. Mountain View, CA: Sun Microsystems, 2001.
- [27] S. Clarke, *Describing and Measuring API Usability with the Cognitive Dimensions*. 2005.
- [28] A. Macvean, M. Maly, and J. Daughtry, "API Design Reviews at Scale," in *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems (CHI EA '16)*, 2016, pp. 849–858.
- [29] J. Stylos and B. A. Myers., "The Implications of Method Placement on API Learnability," in *Sixteenth ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE 2008)*, 2008, pp. 105–112.
- [30] T. Grill, O. Polacek, and M. Tscheligi, "Methods towards API Usability: A Structural Analysis of Usability Problem Categories," in *Human-Centered Software Engineering*, vol. 7623, Winckler, Marco, and E. Al, Eds. Toulouse, France: Springer Berlin Heidelberg, 2012, pp. 164–180.
- [31] U. Farooq, L. Welicki, and D. Zirkler, "API usability peer reviews," in *Proceedings of the 28th international conference on Human factors in computing systems - CHI '10*, 2010.
- [32] M. Henning, "API Design Matters," *ACM Queue*, vol. 5, no. 4, pp. 24–36, 2007.

- [33] I. Rus and M. Lindvall, "Knowledge management in software engineering," *IEEE Softw.*, vol. 19, no. 3, pp. 26–38, 2002.
- [34] E. Bjarnason, K. Smolander, E. Engström, and P. Runeson, "A theory of distances in software engineering," *Information and Software Technology*, vol. 70, pp. 204–219, Feb. 2016.
- [35] H. K. Edwards and V. Sridhar, "Analysis of the effectiveness of global virtual teams in software engineering projects," in *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*, 2003.
- [36] J. Corbin and A. Strauss, "Grounded Theory Research: Procedures, Canons and Evaluative Criteria," *Zeitschrift für Soziologie*, vol. 19, no. 6, p. 515, Jan. 1990.
- [37] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochem. Med.*, vol. 22, no. 3, pp. 276–282, 2012.
- [38] C. Sadowski, J. van Gogh, C. Jaspan, E. Soderberg, and C. Winter, "Tricorder: Building a Program Analysis Ecosystem," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015.
- [39] FxCop. FxCop, 2018.
<https://msdn.microsoft.com/en-us/library/bb429476.aspx>
[Accessed: 2018]
- [40] Clang. Clang-Tidy, 2018.
<http://clang.llvm.org/extr clang-tidy/> [Accessed: 2018]
- [41] E. Murphy-Hill, C. Sadowski, A. Head, J.Daughtry, A.Macvean, C. Jaspan, & C. "Discovering API Usability Problems at Scale." in *Proceedings of the 2nd International Workshop on API Usage and Evolution* (2018), pp. 14-17.
- [42] V. Braun, & V. Clarke. (2006). Using thematic analysis in psychology. *Qualitative research in psychology*, 3(2), 77-101.
- [43] C. Lewis, & J. Rieman. (1993). Task-centered user interface design. *A Practical Introduction*.
- [44] A. J. Ko, & Y. Riche. (2011, September). The role of conceptual knowledge in API usability. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on* (pp. 173-176). IEEE.

DeployGround: A Framework for Streamlined Programming from API Playgrounds to Application Deployment

Jun Kato, Masataka Goto

National Institute of Advanced Industrial Science and Technology (AIST), Tsukuba, Japan, {jun.kato, m.goto}@aist.go.jp

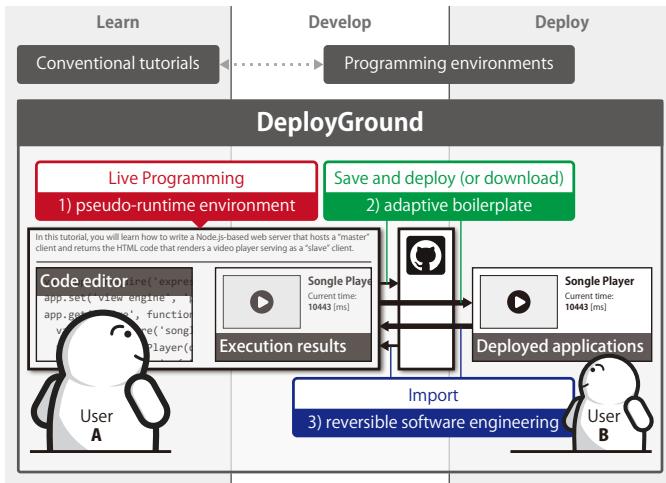


Fig. 1. The DeployGround framework features 1) a *pseudo-runtime environment*, 2) an *adaptive boilerplate*, and 3) a *reversible software engineering* feature for interactive coding tutorials, which altogether streamlines learning APIs on playgrounds and developing and deploying applications.

Abstract—Interactive web pages for learning programming languages and application programming interfaces (APIs), called “*playgrounds*,” allow programmers to run and edit example codes in place. Despite the benefits of this live programming experience, programmers need to leave the playground at some point and restart the development from scratch in their own programming environments. This paper proposes “*DeployGround*,” a framework for creating web-based tutorials that streamlines learning APIs on playgrounds and developing and deploying applications. As a case study, we created a web-based tutorial for browser-based and Node.js-based JavaScript APIs. A preliminary user study found appreciation of the streamlined and social workflow of the DeployGround framework.

Index Terms—Coding tutorials; online learning; API playground; live programming; programming experience

I. INTRODUCTION

It is not easy for programmers to learn new programming languages and application programming interfaces (APIs). Prior research has extensively investigated how to design learnable languages [1] and APIs [2], but only recently has the research community started to discuss the effectiveness of the online learning resources for coding [3], such as interactive tutorials, web references, massive open online courses

(MOOCs), educational games, and creative platforms. Web references and MOOCs courses such as API documentations and step-by-step introductions are often provided in *read-only* formats, in that they consist of text and optional multimedia content, such as images of input and output data and screen recordings of programming environments. To try out the tutorial content, learners need to switch back and forth between the tutorial and their programming environments.

Recent web-based tutorials avoid this frequent context switching by incorporating code editors into web pages, allowing the learners to practice *live programming* with the language or API without installing anything on their computers. A set of such features is often called a “*playground*,” because it constitutes a sandboxed environment in which the learners can play with the target language or libraries (e.g., Khan Academy [4], TypeScript [5], and Vimeo API [6]).

Although the *playground* approach has significant advantages over the conventional *read-only* tutorial, programmers developing applications need to leave the web-based playgrounds and restart the development in their own programming environments. This tedious transition is usually handled by the learners and is not supported by the tutorials. This paper proposes DeployGround, a framework for creating web-based tutorials that streamlines learning APIs on playgrounds and developing and deploying applications (**Figure 1**).

II. RELATED WORK

This section introduces prior work on web-based coding tutorials. More thorough reviews of the related work including research on executable documents [7], [8] and live programming [9]–[13] can be found on the web¹.

While there is much work on creating tutorials for various purposes [14]–[17], there is only a handful of work specialized in creating interactive coding tutorials. Harms et al. explored automatic generation of interactive step-by-step tutorials by transforming each sentence of example codes into a step [18]. Tutoron [19] allows one to write micro-explanations of code and allows learners to read them automatically inserted next to example code on the web. Codepourri [20] allows annotation of the history of program executions through which learners can navigate to learn the program behavior. Our work does not provide tools for creating a new kind of tutorials as these

do but instead presents a framework that addresses limitations of existing web-based coding tutorials for learning APIs.

As discussed in the introduction, many online tutorials present *read-only* content that programmers can read, watch, and sometimes discuss with other learners but cannot interactively run and edit. However, there is an increasing number of interactive coding tutorials that provide code editors with which programmers can run and edit example code. In terms of the implementation, they can be roughly divided into three categories (**Figure 2**).

The first category (**Figure 2 (1)**) is for learning client-side web technologies such as HTML/JavaScript/CSS and utilizes the JavaScript `eval()` function and/or HTML5 sandboxed inline frames (`Iframe`). For instance, W3Schools [21] provide a JavaScript code editor next to the preview pane, in which the code gets executed in the `Iframe`. The `eval()` and `Iframe` implementations are very simple and provide quick response to the user edits, but both are vulnerable to malicious code that can crash the browser (such as infinite loops). Furthermore, this category cannot handle programming languages the browser cannot interpret.

The second category (**Figure 2 (2)**) is for learning how to use the character-based user interface (CUI) and how to build CUI programs. It provides each user access to a lightweight virtual machine (VM) on the server, such as a Docker container. For instance, the C programming course in Tutorials Point [22] provides access to the GCC compiler and allows the user to compile and run the program. Codecademy [23] provides access to a console of a Linux-based VM and allows programmers to test CUI commands. npm [24], the package repository for Node.js libraries, allows the user to test libraries within the browser. Although this approach is flexible and can safely run any code, it is usually slow because of its high computational cost and the latency between the server and client. To make matters worse, all visitors to the web pages need to share the computing resources, which are usually limited owing to the running cost, resulting in even slower responses.

Our work and many live programming environments on the web fall in the third category (**Figure 2 (3)**), in which the user code gets executed in an interpreter. The interpreter is implemented in JavaScript and runs on a web browser. This approach is slightly slower than the `Iframe` method because of the interpreter overhead but significantly faster

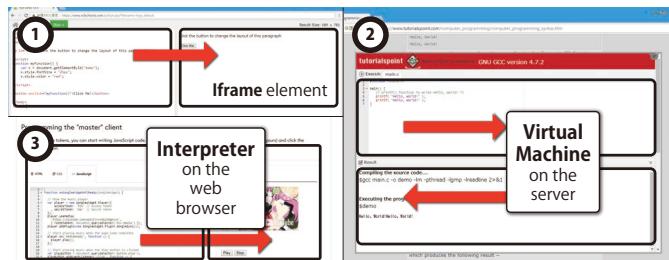


Fig. 2. Three implementation-based categories of interactive coding tutorials.

than the VM-based method because everything runs on the client computer without network or VM overhead. Because the user code is always executed under the supervision of the host interpreter, malicious code can be detected in a practical manner, and it is much safer than the `eval()` and `Iframe` methods. The execution is often more controllable than that in the VM method because there is no black box in the code execution process. Our work implements a *pseudo-runtime environment* that emulates the behavior of a server machine or a microcontroller with a thin interpreter layer and wrapped APIs of a fixed set of libraries.

III. BRIEF REVIEW OF EXISTING TUTORIALS

In this section, we use a JavaScript API called “Songle Sync API [25]” as a representative example of modern APIs and briefly introduce the *previous* version of its web-based tutorial. Then, based on the review and additional analysis of other popular tutorials, we identify four limitations of the existing interactive coding tutorials, each of which contributed to the design of the DeployGround framework.

A. Representative Example API: Songle Sync API

The Songle Sync API allows hundreds of devices to play visual and physical computing performance synchronized with music playback (**Figure 3**). We chose it as a representative example of modern APIs for the following reasons.

- It is provided for JavaScript, which according to the annual report from the social coding platform GitHub was the most popular programming language in 2017 [26].
- Its has been actively developed since its initial release in August 2017.
- It supports both web browsers and physical devices such as the Raspberry Pi [27], reflecting the diverse application domain of modern APIs.
- It involves multiple (sometimes >100) clients over the Internet with real-time communication and handles complex data, making its behavior a non-trivial example of API behavior.
- It provides a large number of methods and properties (73 as of April 2018).

B. Songle Sync API Tutorial

The previous version of the Songle Sync API tutorial links to the API documentation and provides step-by-step explanations of the concepts used in the API. In the later steps, programmers can not only read but also modify the



Fig. 3. Example applications made with Songle Sync API [25]—a web browser-based one (left) and Node.js-based ones (right; actuator modules controlled by Raspberry Pi devices).

example code and try calling the API within the web page that executes code in an `Iframe` element. This is a typical tutorial implementation in the second category (**Figure 2 (1)**), which can be used to prototype a single HTML page containing HTML/JavaScript/CSS code.

C. Limitations of Existing Tutorials

1) *Ephemeral Code*: Example code can be modified, but the modified code is ephemeral. Once the programmer leaves the tutorial, it is gone. The transience of the code prevents learners from continuously growing their codebases and gaining ownership of the code they edit.

Existing tutorials, such as W3Schools, allow one to download the code, but the downloaded code cannot be imported again. DS.js [10] allows the code to be stored in the query parameter of the URL but limits the size of the stored code. Codecademy tutorials and other tutorials that provide a VM instance to each user can keep the session between tutorial steps, but the session cannot be exported to nor imported from a local machine.

2) *Toy Sandbox or Expensive Sandbox*: Existing web-based tutorials tend to suffer from the issues related to the sandbox on which the user code runs. For instance, consider providing a tutorial for building Node.js-based applications. Tutorials simply utilizing `Iframe` or `eval()` do not allow the programmer to edit and test the JavaScript code for the Node.js environment.

Tutorials that use virtual machines, in contrast, can theoretically host the Node.js-based applications. But, running VM instances is computationally (and thus financially) expensive, so many tutorial creators would be unable to provide sufficient computational resources for fluid programming experience. In addition, it is difficult to gain meaningful debugging information when using a virtual machine. Furthermore, there is no way to emulate physical computing devices such as a Raspberry Pi device with a blinking LED.

3) *No Support for Deployment nor Social Interaction*: With many existing web-based tutorials, the learner can download the edited code as a single HTML file. Although the downloaded file can be loaded into a web browser, recent web browsers prohibit executing JavaScript in local files to prevent security risks. There are usually no instructions on how to deploy the code to the HTTP server. Deploying server-side code such as a Node.js-based project is more complex, but typical API tutorials only show text-based instructions or point to external resources that explain how to set up servers.

In addition, the learner needs to learn the content alone. The authors of the tutorial provide example code and nothing more. There is no platform support to collect all of the variations created by previous learners, which could potentially serve as new tutorial content for new learners. Nor is there any way to connect with other learners, who could help the learner with respect to the tutorial content. Social interactions in online learning have been extensively studied in the context of MOOCs as in [28] and [29], but there is not much prior research on how to augment API tutorials with social features.

IV. DEPLOYGROUND FRAMEWORK

We propose the DeployGround framework (**Figure 1**), which addresses the limitations discussed above by revising the interaction design of the existing tutorials. This section provides an overview of the revised Songle Sync API tutorial and explains the key features of the tutorial’s framework.

The revised tutorial website (**Figure 4**) allows the learners to play with APIs, those for prototyping HTML/JavaScript/CSS applications as well as Node.js applications, save project files in GitHub, and deploy the files to public web servers—all without leaving the tutorial website. Additionally, its social features help the user learn from concrete examples.

A. Framework that Covers All Tutorial Steps

The framework provides a *unified workspace* throughout all of the tutorial steps—each code editor in the steps corresponds to a different file in the workspace, and each file can load other files with the `require` function, whose implementation is provided by the *pseudo-runtime environment*. We borrow the concept of the workspace from integrated development environments (IDEs), and in terms of implementation, the tutorials in the DeployGround framework are built on top of the web-based IDE.

With this support for a continuous session throughout the tutorial, we expect the ephemeral code to become permanent, written incrementally by programmers confident of their progress. Unlike the previous version that provided each step almost independently, the revised version makes all steps relevant to each other. For instance, the previous version forced the learner to input string tokens for the API calls in each step, but the revised version makes it possible to create a JavaScript file that is shared among all steps.

B. Pseudo-Runtime Environment

The framework implements a *pseudo-runtime environment* that enables quick execution and debugging of the code written for the deployment target—in the case of the Songle Sync API, a Node.js environment. It is written in a client-side native language (JavaScript for web browsers) and interprets the target language (JavaScript for the Node.js runtime) with

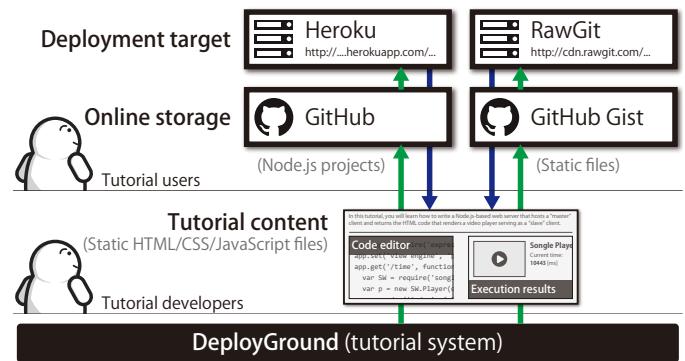


Fig. 4. Overview of the tutorial system implementation, on which tutorial content such as the Songle Sync API tutorial [25] is built.

partial support for the APIs of the default libraries (Node.js libraries such as `fs` for loading local files and `require` for loading `npm` packages).

In the revised tutorial, an emulated web browser or a figure of the Raspberry Pi device is shown next to the editor, both of which render the responses produced by the user code. When the programmer interacts with the emulated browser, the pseudo-runtime environment handles requests to the browser by emulating the execution of the Node.js code. Although the emulation is not perfect, it returns responses almost instantly because there is no network latency and the emulation layer is drastically thinner than that of a VM-based method. We expect the emulation to satisfy the needs of learners quickly experimenting with example code. When unsupported APIs are called, the tutorial shows error messages and a link to the supported APIs. It also shows typical errors such as execution timeout without freezing the browser.

C. Adaptive Boilerplate

When the programmer wants to leave the tutorial and continue the application development, the user code in the tutorial cannot be naively executed in the programmer's environment. Because the framework is in charge of emulating the deployment target, it is aware of the transformation that wraps the user code with some boilerplate. For instance, `package.json` is needed for a Node.js project.

With this support for the adaptive boilerplate, the revised tutorial provides next to every code editor a `download` button that allows the user to download an archive file containing the transformed user code, boilerplate files, and a text file with instructions on how to install an IDE and the Node.js binary and how to run a command (`npm install`) that installs dependent Node.js libraries.

Furthermore, next to the download button is a `deploy` button that deploys the relevant files to the target server. Currently, static files such as HTML/JavaScript/CSS files are saved on a GitHub Gist [30] server and served through its unofficial content delivery service called RawGit [31], and Node.js project files are saved as a GitHub repository and deployed to a PaaS provider called Heroku [32]. After the deployment, the programmer can use a web-based IDE such as Cloud9 [33] to continue the application development.

D. Reversible Software Engineering

The framework facilitates social interactions between learners who visit the tutorials. It utilizes a social coding platform, GitHub, to store the user code. Although it is usually difficult to reverse-engineer deployed web applications, applications developed within the framework can be made *reversible* by design. We call this *reversible software engineering*.

The *adaptive boilerplate* feature in the revised tutorial not only adds the ordinary boilerplate code but also hyperlinks to the tutorial page. By following the hyperlinks, the programmer can start the tutorial from scratch. Additionally, the programmer can optionally import the corresponding GitHub Gist or GitHub repository data into the tutorial. During the loading

process, the project importer strips the boilerplate added by the exporter. The deployed applications thus become a new set of examples from which future learners can benefit.

V. PRELIMINARY USER FEEDBACK

As a preliminary study to gain initial qualitative user feedback, we asked three professional software engineers, two computer science researchers, and twenty-four university students to use the revised Songle Sync API tutorial. We asked the professional engineers and researchers to compare their experience in this use with their prior experience using web-based tutorials, and we asked the students to spend two days using the tutorial and developing applications.

All the participants appreciated the streamlined experience from the playground to deployment. While ordinary web-based tutorials are targeted to a single developer, we observed the university students instantly sharing and boasting about their developed applications, supporting the social aspect of our framework. Other representative insights relevant to future work are discussed below.

1) Potential Applications: While the DeployGround framework has been tested for sandboxing only a web server and Internet of Things devices, there were enthusiastic expectations regarding its potential applications. For instance, the participants requested interactive tutorials for development frameworks for iOS and Android devices and for APIs for machine learning applications.

These expectations stemmed from the high initial cost of trying out the frameworks and APIs. In particular, installing and uninstalling a framework, preparing not only a server but also datasets for testing APIs, and looking for a variety of example codes are tedious.

2) Demands for Architectural Visualizations: A recurring request from the participants was for more explicit visualization of the workflow supported by the DeployGround framework. While the automated project export and import processes were considered extremely helpful, the participants wanted to know more about what is happening behind the scene. In particular, those who did not know the concept of PaaS wanted to see a figure like **Figure 4**, which shows the relationships between the tutorial, GitHub, and Heroku.

3) Limitation and Potential Extension: Given that our approach emulates the target rather than hosting it, there is an inherent limitation that was observed during the user study. For instance, there were complaints about convenient but unsupported APIs. We are aware of such APIs and clearly state in the tutorial that further developments should be done on a web-based integrated development environment, the transition to which should be very smooth thanks to the project exporter feature. Future work should also be done on instantly notifying the users of unsupported APIs—e.g., showing errors when unsupported APIs are typed in the code editor.

ACKNOWLEDGMENT

This work was supported in part by JST ACCEL Grant Number JPMJAC1602, Japan.

REFERENCES

- [1] A. Stefk, S. Hanenberg, M. McKenney, A. Andrews, S. K. Yellanki, and S. Siebert, "What is the Foundation of Evidence of Human Factors Decisions in Language Design? An Empirical Study on Programming Language Workshops," in *Proceedings of the 22nd International Conference on Program Comprehension*, ser. ICPC '14. New York, NY, USA: ACM, 2014, pp. 223–231. [Online]. Available: <http://doi.acm.org/10.1145/2597008.2597154>
- [2] M. P. Robillard, "What Makes APIs Hard to Learn? Answers from Developers," *IEEE Softw.*, vol. 26, no. 6, pp. 27–34, Nov. 2009. [Online]. Available: <http://dx.doi.org/10.1109/MS.2009.193>
- [3] A. S. Kim and A. J. Ko, "A Pedagogical Analysis of Online Coding Tutorials," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '17. New York, NY, USA: ACM, 2017, pp. 321–326. [Online]. Available: <http://doi.acm.org/10.1145/3017680.3017728>
- [4] "Computer Programming — Computing — Khan Academy," accessed April 1, 2018. [Online]. Available: <https://www.khanacademy.org/computing/computer-programming>
- [5] "TypeScript Playground," accessed April 1, 2018. [Online]. Available: <https://www.typescriptlang.org/play>
- [6] "Vimeo API Playground," accessed April 1, 2018. [Online]. Available: <https://developer.vimeo.com/api/playground>
- [7] F. Perez and B. E. Granger, "IPython: A System for Interactive Scientific Computing," *Computing in Science and Engg.*, vol. 9, no. 3, pp. 21–29, May 2007. [Online]. Available: <http://dx.doi.org/10.1109/MCSE.2007.53>
- [8] C. N. Klokmose, J. R. Eagan, S. Baader, W. Mackay, and M. Beaudouin-Lafon, "Webstrates: Shareable Dynamic Media," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '15. New York, NY, USA: ACM, 2015, pp. 280–290. [Online]. Available: <http://doi.acm.org/10.1145/2807442.2807446>
- [9] J. Kato, T. Igarashi, and M. Goto, "Programming with Examples to Develop Data-Intensive User Interfaces," *Computer*, vol. 49, no. 7, pp. 34–42, July 2016.
- [10] X. Zhang and P. J. Guo, "DS.js: Turn Any Webpage into an Example-Centric Live Programming Environment for Learning Data Science," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '17. New York, NY, USA: ACM, 2017.
- [11] J. Kato and M. Goto, "f3.js: A Parametric Design Tool for Physical Computing Devices for Both Interaction Designers and End-users," in *Proceedings of the 2017 Conference on Designing Interactive Systems*, ser. DIS '17. New York, NY, USA: ACM, 2017, pp. 1099–1110. [Online]. Available: <http://doi.acm.org/10.1145/3064663.3064681>
- [12] J. Kato, T. Nakano, and M. Goto, "TextAlive: Integrated Design Environment for Kinetic Typography," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, ser. CHI '15. New York, NY, USA: ACM, 2015, pp. 3403–3412. [Online]. Available: <http://doi.acm.org/10.1145/2702123.2702140>
- [13] C. Roberts, M. Wright, J. Kuchera-Morin, and T. H'ollerer, "Gibber: Abstractions for Creative Multimedia Programming," in *Proceedings of the 22nd ACM International Conference on Multimedia*, ser. MM '14. New York, NY, USA: ACM, 2014, pp. 67–76. [Online]. Available: <http://doi.acm.org/10.1145/2647868.2654949>
- [14] P.-Y. Chi, S. Ahn, A. Ren, M. Dontcheva, W. Li, and B. Hartmann, "MixT: Automatic Generation of Step-by-step Mixed Media Tutorials," in *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '12. New York, NY, USA: ACM, 2012, pp. 93–102. [Online]. Available: <http://doi.acm.org/10.1145/2380116.2380130>
- [15] P.-Y. Chi, J. Liu, J. Linder, M. Dontcheva, W. Li, and B. Hartmann, "DemoCut: Generating Concise Instructional Videos for Physical Demonstrations," in *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '13. New York, NY, USA: ACM, 2013, pp. 141–150. [Online]. Available: <http://doi.acm.org/10.1145/2501988.2502052>
- [16] B. Lafreniere, T. Grossman, and G. Fitzmaurice, "Community Enhanced Tutorials: Improving Tutorials with Multiple Demonstrations," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '13. New York, NY, USA: ACM, 2013, pp. 1779–1788. [Online]. Available: <http://doi.acm.org/10.1145/2470654.2466235>
- [17] K. J. Harms, D. Cosgrove, S. Gray, and C. Kelleher, "Automatically Generating Tutorials to Enable Middle School Children to Learn Programming Independently," in *Proceedings of the 12th International Conference on Interaction Design and Children*, ser. IDC '13. New York, NY, USA: ACM, 2013, pp. 11–19. [Online]. Available: <http://doi.acm.org/10.1145/2485760.2485764>
- [18] A. Head, C. Appachu, M. A. Hearst, and B. Hartmann, "Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code," in *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct 2015, pp. 3–12.
- [19] M. Gordon and P. J. Guo, "Codepourri: Creating Visual Coding Tutorials Using a Volunteer Crowd of Learners," in *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct 2015, pp. 13–21.
- [20] "W3Schools Online Web Tutorials," accessed April 1, 2018. [Online]. Available: <https://www.w3schools.com>
- [21] "Tutorials Point," accessed April 1, 2018. [Online]. Available: <https://www.tutorialspoint.com>
- [22] "Codecademy," accessed April 1, 2018. [Online]. Available: <https://www.codecademy.com>
- [23] "npm," accessed April 1, 2018. [Online]. Available: <https://www.npmjs.com/>
- [24] "Songle Sync Tutorial," accessed April 1, 2018. [Online]. Available: <http://tutorial.songle.jp-sync>
- [25] "GitHub Octoverse 2017," accessed April 1, 2018. [Online]. Available: <https://octoverse.github.com/>
- [26] "Raspberry Pi," accessed April 1, 2018. [Online]. Available: <https://www.raspberrypi.org/>
- [27] J. Kay, P. Reimann, E. Diebold, and B. Kummerfeld, "MOOCs: So Many Learners, So Much Potential ..." *IEEE Intelligent Systems*, vol. 28, no. 3, pp. 70–77, May 2013.
- [28] C. G. Brinton, M. Chiang, S. Jain, H. Lam, Z. Liu, and F. M. F. Wong, "Learning about Social Learning in MOOCs: From Statistical Analysis to Generative Model," *IEEE Transactions on Learning Technologies*, vol. 7, no. 4, pp. 346–359, Oct 2014.
- [29] "GitHub Gist," accessed April 1, 2018. [Online]. Available: <https://gist.github.com>
- [30] "RawGit," accessed April 1, 2018. [Online]. Available: <https://rawgit.com>
- [31] "Heroku," accessed April 1, 2018. [Online]. Available: <https://www.heroku.com>
- [32] "Cloud9," accessed April 1, 2018. [Online]. Available: <https://ide.c9.io>

Human-AI Interaction in Symbolic Problem Solving

Benjamin T. Jones

Paul G. Allen School of Computer Sci. and Engr.

University of Washington

Seattle, Washington 98195

Email: benjones@cs.washington.edu

Abstract—Despite the increasing need for computer assistance in solving problems involving complex systems and large amount of data, professional mathematicians, scientists, and engineers currently avoid the use of computer algebra systems during creative problem-solving phases of their work due to problems with transparency, familiarity, and inflexibility in input. I have designed and prototyped a new approach to interaction with computer algebra systems that is compatible with current working styles, flexible in its input and output. I propose a user study to validate this tool, and tool extensions to allow creative problem solvers to interactively define their own notation as they work.

I. MOTIVATION

Symbolic reasoning is a crucial task for many scientists, engineers and mathematicians. As the complexity of models and the size of data sets used in these fields has grown, there is an ever increasing need for computer assistance in tackling these problems. Computer algebra systems (CAS) designed to work on these problems have existed for decades, but they are not commonly used for creative problem solving.

A series of interviews and observational studies between 2009 and 2013 found that scientists, mathematicians, and engineers avoid using computer algebra tools in creative problem solving. Those professionals who did use them tend to reserve CAS usage for verifying previously hand-done calculations, or programmatically exhausting large search spaces for known patterns. The reasons for avoiding CAS are a lack of 2d (traditional hand-written) notation for input, a lack of transparency of operations to build trust in the results, and most importantly a rigid input format that is difficult to iterate on quickly and accurately, and that constrains creative thought to expressions easily expressible in the input notation [1].

Prior advances in mathematical tooling have enabled more creative problem solving by mathematicians and others by abstracting away low-level concerns and allowing for the development of human intuition at higher levels. One example is the adoption of algebraic notation, which enabled the development of modern analytic calculus and geometry by allowing intuition about symbolic manipulation to proxy for physical objects and word problem. Another is modern vector notation, which abstracted large systems into intuitible single equations. This invention spurred and explosion of progress in physics as systems that were previously too complex to study holistically were made intuitible; the birth of modern physics traces back to these developments [2].

The problems at the forefront of STEM fields today deal with systems of equations an order of magnitude more complex than those made tractable by these previous notational advances, and also with vast quantities of data which are infeasible for any human to review by themselves, necessitating computer assistance both to perform calculations and to understand the data involved. But our best computer tools for dealing with these types of systems are going unused exactly at the stage of mathematical invention where intuition and new generalizations are likely to be discovered: the ideation phase. I believe that we need a new way for humans to interact with symbolic reasoning that merges human intuition for problem solving with powerful computation.

Ideally, such a system would, like previous developments, extend upon existing notational technology to benefit from the expertise of existing users and to inherit the affordances useful in current workflows. I propose a digital ink interface to computer algebra systems that uses traditional handwritten derivations as its input. Users would explore symbolic systems using familiar notation and techniques, but the underlying computer algebra system would allow them to dexterously manipulate far more complex and data-backed expressions by automatically completing and correcting existing derivations as the user writes, and suggesting further manipulations. This interface would not suffer from the oppressive rigidity of current systems both because it supports familiar input, but also because its design does not impose notational conventions, and even allows the invention of new notation on-the-fly.

I have built a prototype of the algebraic completion and suggestion piece of this system. Here I propose a user study to validate this prototype, as well as extending the prototype into a system usable by professionals via interface improvements, making it agnostic to notational conventions, and extensible to new notations and other fields using symbolic notations.

II. BACKGROUND

A. Problem Solving Formalism

The classical formalism for artificial intelligence as proposed by Simon models problem solving as a state-space search, where each state is a potential solution, and operations transform one potential solution into another [3]. These states and solutions form a graph, which an AI agent explores by building a search tree. An agent can find the desired solution by matching a search criteria or optimizing a metric over solution states.

Symbolic manipulation fits neatly into this model. States are algebraic expressions and equations, and the rules of algebra (or another formal system) define the operations. Traditional derivation puts humans in the position of AI agents, manually applying operators (an error prone process, especially as expressions become complex), and using experience and intuition to plan their search. In an exploratory context, human solvers will often not have a concrete goal in mind – their goal is to better understand the system in question through manipulation, or to discover interesting and useful identities.

B. Computer Algebra Systems

Computer algebra systems like Mathematica and Maple follow this model. They provide two types of functions – applications of particular transformations (functions such as factor or expand), and AI search functions that apply many operations heuristically in search of a particular goal or optimization (solve or simplify). As Bunt et. al. illuminated, operating in the former mode is too clunky and restrictive to be useful in creative contexts, and the later method is unsuitable for exploratory contexts because the user does not have or cannot articulate a concrete goal.

C. Natural Input

Other systems have attempted to make CAS more usable by offering a handwriting frontend. Mathbrush allowed initial expression input via tablet and presented individual CAS operations via dropdown and context menus [4]. Hands-on-Math added manipulation via gestures and demonstrated that these natural input techniques increase ease-of-use for the operations they permit [5]. To date, no such system has gained widespread use, lacking the full power of a general CAS [1].

D. Searching Over Search Trees

In recent work, I designed a framework for building symbolic manipulation interfaces that addresses the major concerns professionals currently have with CAS [6]. The key insight is that issuing commands to a symbolic solver requires interpretation of those commands, leading both to a restriction in valid forms of input and to implementation overhead on the scale of the number of commands. Rather than interpret input as commands for the solver, my solver continuously performs a search in the background, caching its search tree. User input in a traditional derivation is then interpreted as intermediate goals in the form of queries.

These queries are for states that are symbolically similar in form. This way the only interpretation that is required is accurate math handwriting recognition, which exists for complex expressions [7]. Symbolic similarity is matched against all notational variations of a particular state (e.g. $\sum i$ and $1 + 2 + \dots$ both represent the same expression, and are often used simultaneously in derivations), which obviates the problem of restricting the user to one notational convention.

Since this system builds off existing CAS software, it inherits the computational power of those systems, as well as their ability to define custom notations. This allows the

system to be extended to other symbolic systems (chemical formulas, for example), without the burden of any additional UI programming. Anyone capable of expressing their notation in LaTeX syntax can extend the notations usable by the system.

III. PROPOSED WORK

I will conduct a user study to evaluate and validate this prototype. The current design is based on my previous experience in applied mathematics research, so it is possible that the distance metrics used to match queries are biased towards my views of mathematics.

The proposed study has two components. The first is a predictive task in which participants are shown a query and a collection of potential results, and asked to predict how the system will rank them. Another predictive task will not present potential results, but ask the participants what they would expect the top result to be. These tasks are intended to determine if the query results minimize surprise on the part of the user, which is crucial if users are to develop intuition for working with the tool.

The second type of task is a usability evaluation. Participants will be given a starting expression and a goal to reach using the system (e.g. isolate an expression or solve for a variable), and evaluated on the number of steps and amount of backtracking required to reach the goal. This will test if my interaction technique is usable for computer algebra. Observations of this task will also help guide future improvements.

Several improvements to the prototype are currently being implemented. The current CAS integration does not expand expressions into all notational variants adding a module to do so will make querying within the system highly flexible.

Adding new notations currently requires skill in programming the underlying computer algebra system. I plan to add a meta-notation that allows the definition of new notation in LaTeX syntax and pushing it down to the underlying solver.

Finally, I intend to integrate my prototype into a whiteboard style digital inking interface to allow it to be used within existing pen-and-ink exploration workflows.

REFERENCES

- [1] A. Bunt, M. Terry, and E. Lank, “Challenges and Opportunities for Mathematics Software in Expert Problem Solving,” *Human-Computer Interaction*, vol. 28, pp. 222–264, May 2013.
- [2] V. Katz, *A History of Mathematics*. Pearson/Addison-Wesley, 2004.
- [3] H. A. Simon, *The sciences of the artificial*. MIT press, 1996.
- [4] G. Labahn, E. Lank, S. MacLean, M. Marzouk, and D. Tausky, “Mathbrush: A system for doing math on pen-based devices,” in *Document Analysis Systems, 2008. DAS’08. The Eighth IAPR International Workshop on*, pp. 599–606, IEEE, 2008.
- [5] R. Zeleznik, A. Bragdon, F. Adeputra, and H.-S. Ko, “Hands-on Math: A Page-based Multi-touch and Pen Desktop for Technical Work and Problem Solving,” in *Proceedings of the 23rd Annual ACM Symposium on User Interface Software and Technology, UIST ’10*, (New York, NY, USA), pp. 17–26, ACM, 2010.
- [6] B. T. Jones and S. L. Tanimoto, “Searching Over Search Trees for Human-AI Collaboration in Exploratory Problem Solving: A Case Study in Algebra,” in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, (in press), Oct. 2018.
- [7] E. M. Taranta, A. N. Vargas, S. P. Compton, and J. J. Laviola, “A Dynamic Pen-Based Interface for Writing and Editing Complex Mathematical Expressions With Math Boxes,” *ACM Transactions on Interactive Intelligent Systems*, vol. 6, pp. 1–25, July 2016.

Supporting Effective Strategies for Resolving Vulnerabilities Reported by Static Analysis Tools

Justin Smith

Department of Computer Science
North Carolina State University
Raleigh, North Carolina 27606

Email: jssmit11@ncsu.edu

Abstract—Static analysis tools detect potentially costly security defects early in the software development process. However, these defects can be difficult for developers to accurately and efficiently resolve. The goal of this work is to understand the vulnerability resolution process so that we can build tools that support more effective strategies for resolving vulnerabilities. In this work, I study developers as they resolve security vulnerabilities to identify their information needs and current strategies. Next, I study existing tools to understand how they support developers' strategies. Finally, I plan to demonstrate how strategy-aware tools can help developers resolve security vulnerabilities more accurately and efficiently.

I. INTRODUCTION

Static analysis tools, like Coverity [1] and Findbugs [2], enable developers to detect security vulnerabilities early in development. These tools locate and report on potential software security vulnerabilities, such as SQL injection and cross-site scripting, even before code executes. Detecting these defects early is important, because long-linging defects may be more expensive to fix [3]. According to a recent survey by Christakis and colleagues, developers seem to recognize the importance of detecting security vulnerabilities with static analysis; among several types of code quality issues, developers rank security issues as the highest priority for static analysis tools to detect [4].

To actually make software more secure, however, static analysis tools must go beyond simply detecting vulnerabilities. These tools must be usable and enable developers to resolve the vulnerabilities they detect. As Chess and McGraw argue, “Good static analysis tools must be easy to use, even for non-security people. This means that their results must be understandable to normal developers who might not know much about security and that they educate their users about good programming practice” [5].

Unfortunately, evidence suggests existing tools are not easy for developers to use. Researchers cite several related reasons why these tools do not help developers resolve defects, for instance, the tools: produce “bad warning messages” [4] and “miscommunicate” with developers [6]. As a result, developers make mistakes and need help resolving security vulnerabilities due to the poor usability of security tools [7].

Recently, Acar and colleagues introduced a research agenda for improving the usability of security tools, explaining that “Usable security for developers has been a critically under-investigated area” [8]. The goal of this thesis is to investigate and improve the usability of security-oriented static analysis tools so that we can ultimately enable developers to create more secure software.

II. VULNERABILITY RESOLUTION STRATEGIES

My thesis studies the usability of security-oriented static analysis tools through the lens of *vulnerability resolution strategies*. Building on Bhavani and John’s definition of a strategy [9], we define a vulnerability resolution strategy as: a developers’ method of task decomposition that is non-obligatory and directed toward the goal of resolving a security vulnerability. My thesis argues that tools can better help developers resolve vulnerabilities by presenting effective *vulnerability resolution strategies* alongside the defects they detect.

III. USABILITY OF STATIC ANALYSIS

Outside the domain of security, researchers have studied the human aspects of using static analysis tools to identify and resolve defects. Muske and Serebrenik survey 79 studies that describe approaches for handling static analysis alarms [10]. They organize existing approaches into seven categories, which include “Static-dynamic analysis combinations” and “Clustering.” Sadowski and colleagues [11] report on the usability of their static analysis ecosystem at Google, Tricorder. Their experiences suggest that warnings should be easy to understand and fixes should be clear, which motivates the work in this thesis. Similarly, Ayewah and colleagues describe their experiences running static analysis on large code bases. They make suggestions for how tools should help developers triage the numerous warnings that initially might be reported [12]. In comparison, our work focuses on how developers resolve individual security vulnerabilities.

IV. EVALUATION PLAN

Phase 1 (Complete): What information do developers need while using static analysis tools to diagnose potential security vulnerabilities? To understand developers’ information needs while using a security-oriented static analysis tool,

I conducted a think-aloud study with ten participants [13]. I observed participants as they assessed four potential security vulnerabilities using Find Security Bugs [14], a security extension of FindBugs [2]. To identify information needs, a collaborator and I coded transcriptions from participants' audio recordings. To identify emergent categories in the information needs, we conducted an open card sort. Our card sort was validated by two external researchers, who substantially agreed with our categorization ($\kappa = .63$ and $\kappa = .70$, respectively). This study provides us with an initial framework to understand the vulnerability resolution process.

Phase 2 (Complete): What are developers' strategies for acquiring the information they need? We were motivated to extend our prior information needs study, because we wanted to understand how developers *answered*, or failed to answer, their questions. In this follow-up work we explored how developers acquire the information they need through strategies. To answer this second research question, we reanalyzed the data collected from the Phase 1 study to identify strategies [15].

Phase 3 (In Progress): How do existing static analysis tools support developers' information needs and strategies? During Phase 1 and Phase 2, we studied aspects of *developers' behavior* while interacting with a single security-oriented static analysis tool. To answer RQ3 we shift focus from the developer onto the tools, studying how characteristics of existing analysis tools contribute to and detract from the vulnerability resolution process.

We have conducted a heuristic walkthrough evaluation [16] of three open source security tools and plan to extend the evaluation to include commercial tools. As a result of our heuristic walkthrough evaluation so far, we have identified a list of 155 usability issues. We are also in the process of conducting interviews with security experts about their use of static analysis tools. Together, these studies will inform the design of a new static analysis tool interface (Phase 4).

Phase 4 (Proposed): How can we design tools that support more accurate and efficient resolution strategies? To answer this fourth research question I will demonstrate, through novel tool design, how we can apply our findings from the previous three research questions. Particularly, I will create a tool that explicitly supports more effective vulnerability resolution strategies. Figure 1 depicts a mockup of the tool I will build. Its interface reifies effective strategies in hierarchically structured checklists that can be executed by developers who would otherwise lack strategic knowledge.

I hypothesize that such tool will be most beneficial for novice developers, since security experts might have already internalized effective strategies. Therefore, I plan to evaluate this tool in an educational setting among students with relatively little exposure to secure software development. To measure accuracy and efficiency, we will record the number of vulnerabilities participants resolve with the new tool and how long they spend resolving each vulnerability and compare their performance against a baseline. I will triangulate these measures by also capturing usability metrics.

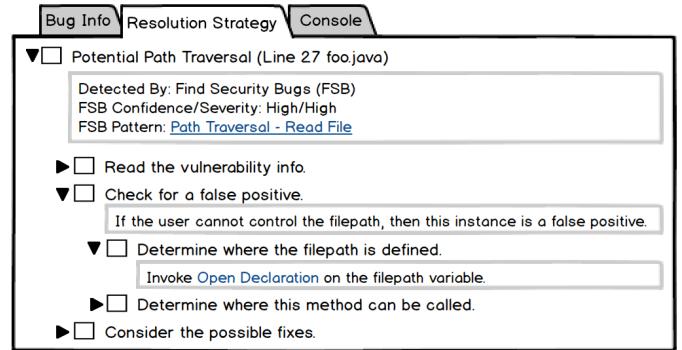


Fig. 1: A mockup of a tool that presents successful strategies.

V. ACKNOWLEDGMENTS

I owe thanks to my advisor, Dr. Emerson Murphy-Hill, my dissertation committee, and the many collaborators who have contributed to this work. This material is based upon work supported by the National Science Foundation under grant number 1318323.

REFERENCES

- [1] "Coverity home page," <https://scan.coverity.com/>, 2018.
- [2] "Findbugs," <http://findbugs.sourceforge.net>.
- [3] R. S. Pressman, *Software engineering: a practitioner's approach*. Palgrave Macmillan, 2005.
- [4] M. Christakis and C. Bird, "What developers want and need from program analysis: An empirical study," in *IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 332–343.
- [5] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security and Privacy*, vol. 2, no. 6, pp. 76–79, Nov. 2004.
- [6] B. Johnson, R. Pandita, J. Smith, D. Ford, S. Elder, E. Murphy-Hill, S. Heckman, and C. Sadowski, "A cross-tool communication study on program analysis tool notifications," in *International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 73–84.
- [7] M. Green and M. Smith, "Developers are not the enemy!: The need for usable security apis," *IEEE Security and Privacy*, vol. 14, no. 5, pp. 40–46, 2016.
- [8] Y. Acar, S. Fahl, and M. L. Mazurek, "You are not your developer, either: A research agenda for usable security and privacy research beyond end users," in *IEEE SecDev*. IEEE, 2016, pp. 3–8.
- [9] S. K. Bhavnani and B. E. John, "The strategic use of complex computer systems," *Human-Computer Interaction*, vol. 15, no. 2, pp. 107–137, Sep. 2000.
- [10] T. Muske and A. Serebrenik, "Survey of approaches for handling static analysis alarms," in *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2016, pp. 157–166.
- [11] C. Sadowski, J. Van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *IEEE International Conference on Software Engineering*. IEEE Press, 2015, pp. 598–608.
- [12] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *ACM Workshop on Program Analysis for Software Tools and Engineering*. ACM, 2007, pp. 1–8.
- [13] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford, "Questions developers ask while diagnosing potential security vulnerabilities with static analysis," in *ACM International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE 2015. ACM, 2015, pp. 248–259.
- [14] "Find security bugs," <http://h3xstream.github.io/find-sec-bugs/>.
- [15] J. Smith, B. Johnson, E. Murphy-Hill, B.-T. Chu, and H. Richter, "How developers diagnose potential security vulnerabilities with a static analysis tool," *IEEE Transactions on Software Engineering*, 2018.
- [16] A. Sears, "Heuristic walkthroughs: Finding the problems without the noise," *Human-Computer Interaction*, vol. 9, no. 3, pp. 213–234, 1997.

The novice programmer needs a plan

Kathryn Cunningham

School of Information, University of Michigan

kicunn@umich.edu

I. INTRODUCTION

Algorithms and automation run social worlds, support scientific discovery, and even arbitrate economic opportunity. Job opportunities in computer science match this outsized influence: projected job growth in computing dwarfs that of other STEM fields [1]. In recognition of this reality, the movement to expand computing education to *all* students, including low-income, underrepresented minority, and female students, has grown by leaps and bounds. This has led to computing instruction in K-12, more computing in colleges, and a more diverse set of students to teach.

However, current approaches to teaching programming often fall short. Multi-institutional, multi-national studies have shown that many students complete a college-level introductory computing course without being able to write basic programs [2], or in some cases, read small pieces of code and predict their result [3]. Extending current teaching techniques into earlier grades or to groups with weaker academic preparation isn't a promising approach.

Something isn't working in introductory computing classrooms—and the dominant strategy for programming instruction should be re-examined. A typical programming course today focuses on *syntax elements* as units of programming knowledge, with textbook chapters often arranged to cover one or two elements (e.g. if/else statements) at a time. Similarly, validated assessments of introductory programming knowledge, like the FCS1 [4], have conceptual topics that read like the Backus-Naur form of a programming language grammar: "Logical Operators", "Assignment", "Definite Loop (for)", etc.

While a detailed focus on the behavior of syntax elements has been shown to improve student outcomes (e.g. [5]), focusing on this behavior alone has drawbacks. There are endless ways to combine syntax elements, and a correspondingly large mental search space for the novice programmer to think through when writing code or understanding someone else's code. Familiarity with how coding syntax works doesn't directly explain *how* to build a program that achieves a certain goal or immediately illuminate *why* someone wrote code the way they did. An instructional technique that streamlines this process may lead more students to success.

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGC-1148903

978-1-5386-4235-1/18/\$31.00 ©2018 IEEE

II. A PLAN-BASED APPROACH HOLDS PROMISE

In contrast to a syntax focus, some computing education researchers have explored an explicit focus on common *patterns* or *strategies* used when coding. While rare in classrooms, this approach potentially aligns with a basic psychological fact: humans often use *schemas* (mental patterns or frames) to organize their knowledge. If we can teach schemas that correspond to common patterns used by programmers, we potentially provide students with a powerful problem-solving tool.

In the 1980s, Elliot Soloway developed a framework of goals and plans to explain how programs are written. A programmer has *goals* they want to achieve and corresponding *plans* of code that achieve the goals. These plans are "canned" solutions to common programming problems, such as checking if an input is valid, summing up all values in a list, or stopping a search when a sentinel value is read. Soloway criticized the syntactic approach as avoiding the parts of programming where students really struggle: composing pieces of a program together into a functional whole [6].

III. STUDENTS PROBLEM-SOLVE THROUGH PATTERNS

We know that experts use plans to problem-solve while writing code [7], but is this approach also appropriate for novices? My recent work suggests that use of programming plans is natural even for students in their first programming course.

I interviewed 13 students about the ways they sketched and drew on scratch paper while solving problems on a recent introductory programming exam. I found that tracing through the behavior of a piece of code was often performed in service of a search for some sort of structure or pattern with which to organize their thoughts. For example, one student described her success in determining the goal of a code snippet she was asked to analyze on the exam:

"I was just writing out just to see which numbers I was going to deal with. Then afterwards I would just look at it and I would be like, oh, so it's going to look for the one that's the greatest..."

A careful trace on scratch paper was also used to confirm the plan or pattern a student had tentatively identified:

"I saw the pattern, and I just wanted to write it up to make sure the pattern was right."

Rather than simply mimicking each step of code execution, these students are taking the very human action of searching for meaning in which to ground their problem-solving. In code reading problems, we know that tracing the behavior of

code execution on paper is correlated with greater problem-solving success [3]. However, in practice, students searched for what the code was “supposed” to do, and turned to careful tracing only when the goal of the code was unclear or needed confirmation. It seems that the ability to accurately recognize plans may mediate the success of novice programmers.

IV. WORKING WITH STUDENT TENDENCIES, NOT AGAINST

Students’ search for plans and patterns is a potentially useful tendency that could be strengthened as a problem-solving approach. There is an opportunity for instructors to facilitate use of this strategy for all students, but little work has been done about how to explicitly incorporate plans into an instructional approach.

Instead, much recent work in building student programming skill has focused on improving understanding of program behavior. Many iterations of program visualization tools have been created to demonstrate the changes in memory as code executes [9]. These tools execute every step of code execution, giving a look “inside” the computer. However, they do not have the capability to infer the goal of the code they visualize, or even any patterns or structure in the changes of key variables.

V. FUTURE DIRECTIONS

The focus on plans as an alternative instructional approach is promising, but much work is needed in order to prove its effectiveness and make it actionable in the classroom. While past work has described how students may build programs by composing goals and plans, little work has focused on a more foundational skill: the ability to *recognize* plans in practice. My prior work has shown that this recognition is key to students’ use of plans while solving problems. I plan to investigate the following research questions:

Is the ability to recognize and recall programming plans correlated with problem-solving success?

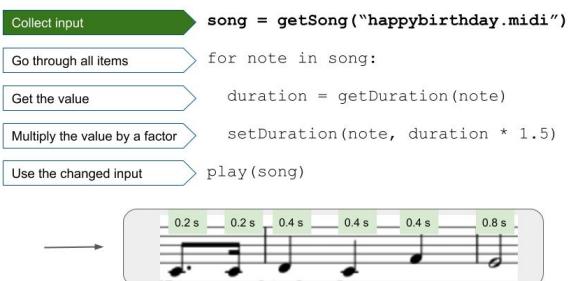
A closer investigation of how knowledge of programming plans is related to success in different types of programming problems, such as reading code, fixing code, and writing code, will help us understand the promise and limitations of this approach.

How can instructors increase the ability of students to recognize and recall programming plans?

There are two techniques I plan to investigate. The first approach is to use examples of the plan implemented in a variety of contexts (see Figure 1). This may allow each student to create a mental schema of the abstracted plan that fits within their existing knowledge structures. The second approach is explicit instruction about a programming plan, using a visualization of an abstract plan. While this approach has the potential advantage of more accurately sharing expert knowledge and decreasing the opportunity for misconceptions, it also has the potential downside of being too abstract or difficult to understand for novices.

Understanding the utility of plan recognition for novices and validating techniques for building plan knowledge add a multipurpose tool to the toolbox of programming educators.

Goal: Increase the duration of all notes, using a factor of 1.5



Goal: Decrease the price of all products, using a factor of 0.75



Fig. 1. Examples of a plan in two different contexts.

REFERENCES

- [1] S. Fayer, A. Lacey, and A. Watson. “BLS spotlight on statistics: STEM occupations - past, present, and future,” Bureau of Labor Statistics, Jan. 2017. Available: <https://www.bls.gov/spotlight/2017/science-technology-engineering-and-mathematics-stem-occupations-past-present-and-future>
- [2] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. Ben-David Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz. “A Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-year CS Students,” In Working group reports from Innovation and Technology in Computer Science Education, 2001, pp. 125-180.
- [3] R. Lister, O. Seppala, B. Simon, L. Thomas, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Mostrom, and K. Sanders. “A multi-national study of reading and tracing skills in novice programmers,” In Working group reports from Innovation and Technology in Computer Science Education, 2004, pp. 119-150.
- [4] A. E. Tew and M. Guzdial. “Developing a validated assessment of fundamental CS1 concepts,” In Proceedings of the 41st ACM Technical Symposium on Computer Science Education, 2010, pp. 97-101.
- [5] G.L. Nelson, B. Xie, and A. J. Ko. “Comprehension first: evaluating a novel pedagogy and tutoring system for program tracing in CS1,” In Proceedings of the 2017 ACM Conference on International Computing Education Research, 2017, pp. 2-11.
- [6] J.C. Spohrer, E. Soloway, and E. Pope. “A goal/plan analysis of buggy Pascal programs,” *Human-Computer Interaction*, vol. 1, no. 2, Jun., pp. 163-207, 1985.
- [7] R. S. Rist. “Schema creation in programming,” *Cognitive Science*, vol. 13, pp. 389-414, 1989.
- [8] J. Sorva, “Notional machines and introductory programming education,” *Transactions on Computing Education*, vol. 13, no. 2, Jul., pp. 1-31, 2013.
- [9] J. Sorva, V. Karavirta, and L. Malmi. “A review of generic program visualization systems for introductory programming education,” *Transactions on Computing Education*, vol. 13, no. 4, Nov., pp. 15-78, 2013.
- [10] M. De Raadt, R. Watson, and M. Tolman. “Teaching and assessing programming strategies explicitly,” In Proceedings of the Eleventh Australasian Conference on Computing Education, 2009, pp. 45-54.

Using Program Analysis to Improve API Learnability

Kyle Thayer

*Paul G. Allen School of
Computer Science & Engineering
University of Washington
Seattle, WA, USA
kthayer@cs.washington.edu*

Abstract—Learning from API documentation and tutorials is challenging for many programmers. Improving the learnability of APIs can reduce this barrier, especially for new programmers. We will use the tools of program analysis to extract key concepts and learning dependencies from API source code, API documentation, open source code, and other online sources of information on APIs. With this information we will generate learning maps for any user-provided code snippet, and will take users through each concept used in the code snippet. Users may also navigate through the most commonly used features of an API without providing a code snippet. We also hope to extend this work to help users find the features of an API they need and also help them integrate that into their code.

Index Terms—API learnability, program analysis, auto-generated documentation

I. BACKGROUND AND MOTIVATION

Programmers make regular use of the many Application Programming Interfaces (APIs) as they write their software. Using APIs and developing strategies to learn APIs can be challenging [1]. These challenges include deciding what the programmer want the computer to do regardless of the programming language or library, to the specific challenges of selecting programming interfaces, knowing how they work, and knowing the relevant concepts and terminology [2], [3]. Using API documentation provides specific challenges as well, such as navigating documentation, understanding how API designers intended their APIs to be used, matching specific scenario needs with API features [4]. To make good decisions on what API features to use and use them properly, programmers need to learn the API they are working with.

Existing methods of learning many large APIs consist of formatted code documentation (i.e., JavaDocs), sometimes with examples and a brief intro, and human-created tutorials. The raw documentation is often difficult for newcomers to navigate and the human-created tutorials take large amounts of effort and can go out of date when new versions of APIs are released. Our previous study on coding bootcamps showed the challenge of learning APIs and how the ability to learn from API documentation and other resources are seen as a valuable skill that was difficult to acquire [1].

Researchers have proposed generating on-demand documentation [5] and have made various attempts to make APIs

easier to learn and use. These attempts include changes to APIs designs [6], and improved methods of searching API documentation [7] and other online resources [8]. These methods focus on improving the search for specific desired features instead of on explaining user-provided code snippets.

II. RESEARCH GOALS & METHODS

We have developed a theory of API knowledge (not yet submitted for publication) which lays out the types of knowledge needed to learn and use APIs. Our future work will take this theory and then build off of previous work of others (e.g., automatically extracting example code [9], automatically extracting input and output information [10]) to automatically extract all parts of this knowledge from API source code, documentation and examples. We will create *learning maps* for APIs which will be constructed from of *key concepts* and *learning dependencies*. By *key concepts* we mean the terminology, ideas, and patterns needed to perform tasks with an API, and by *learning dependencies* we mean the ways in which some *key concepts* can only be understood in relation to other *key concepts*.

These *learning maps* can be used by programmers wanting to understand code they've found or written and see how it can be expanded. They can also be used by tutorial creators in organizing their tutorials (saving time) or by newcomers as a guide for which concepts to learn (giving guidance). In particular, we believe newcomers will benefit from seeing a concise layout of *key concepts* which they can compare against their prior knowledge. The *learning maps* will provide paths to learning any *key concept* in an API. This will support learning one, some, or all features of the API.

In our research we will ask and attempt to answer the following research questions about *key concepts*, *learning dependencies*, and *learning maps*:

- 1) Can we extract *key concepts* and *learning dependencies* from available API code, documentation, open source repositories and question and answers sites?

To extract *key concepts* and *learning dependencies* from available code, we will first look at multiple APIs and tutorials. We will determine from the content and organization of tutorials what each one considers the *key concepts* of an API and what order those concepts can be presented in. These may not be

the only *key concepts* a learner may need to know, but they will provide a baseline of concepts and APIs we will hope to recover through automated methods.

We will look at how *key concepts* and *learning dependencies* might be extracted from existing code, and other online resources, but most of the work in clarifying how to extract them will be done in conjunction with answering the next question.

Additionally, since there are many APIs, APIs change quickly, and new APIs are created, we want to use automated processes to do this work, so our second research question is:

- 2) Can we use program analysis to automatically extract *key concepts* and *learning dependencies* for an API?

Program analysis allows the automated extraction of features from computer programs, whether from code, execution information, or other resources. These analyses often provide information about how programs are expected to work, such as profiling, performance evaluation and bug detection [11]. We will instead use program analysis to identify *key concepts* and *learning dependencies* in APIs.

We will take the example APIs we looked at before and turn to what available code we can find for those APIs, such as: the API code, API documentation, open source code that uses the API and other resources on the API such as StackOverflow. We will then create definitions of *key concepts* and *learning dependencies* in terms of this available code and create program analyses that can extract them. As we iterate through this process we will come up with clearer definitions of our terms and better methods of automatically extracting these features.

- 3) How effective are *learning maps* generated from our automatically extracted *key concepts* and *learning dependencies*?

To test this, we will create an interface for learners that will give them a generated *learning map*. This interface may include automatically generated links to content for learning each concept or manually curated links for the concepts. We will then give learners tasks to complete with an API with the generated *learning map*, measure the effectiveness of these *learning maps* in terms of conceptual learning, problem solving ability, and perceived difficulty. Through this we hope to gain insights for improving to the underlying algorithms and the presentation of these *learning maps*.

- 4) Can we make *learning maps* based on provided code using an API?

To do this, detect which parts of an API are being used in the code and find the relevant sections of the *learning map*. We will then generate a tutorial that highlights concepts used in their code and also suggests possible extensions to the code.

- 5) Can we help developers search *learning maps* for the *key concepts* they need?

To do this, we will take user inputted searches and code, and find *key concepts* related to their input and their code while also considering how commonly used those concepts are. We can then help them integrate the *key concepts* into the code.

III. EXPECTED CONTRIBUTIONS

When this work has been completed, we will have created new analyses that extract newly defined factors from code bases: *key concepts* and *learning dependencies*. We will also have used these analyses to create new content and tools for API learners with specific code questions, newcomers to an API, and creators of API tutorials. Finally, we will have gained new understanding in how programmers learn APIs and what their needs are.

REFERENCES

- [1] K. Thayer and A. J. Ko, "Barriers Faced by Coding Bootcamp Students," in *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ser. ICER '17. New York, NY, USA: ACM, 2017, pp. 245–253. [Online]. Available: <http://doi.acm.org/10.1145/3105726.3106176>
- [2] A. Ko, B. Myers, and H. Aung, "Six Learning Barriers in End-User Programming Systems," in *2004 IEEE Symposium on Visual Languages and Human Centric Computing*, Sep. 2004, pp. 199–206.
- [3] A. J. Ko and Y. Riche, "The role of conceptual knowledge in API usability," in *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Sep. 2011, pp. 173–176.
- [4] M. P. Robillard and R. DeLine, "A field study of API learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, Dec. 2011. [Online]. Available: <https://link.springer.com/article/10.1007/s10664-010-9150-8>
- [5] M. P. Robillard, A. Marcus, C. Treude, G. Bavota, O. Chaparro, N. Ernst, M. A. Gerosa, M. Godfrey, M. Lanza, M. Linares-Vsquez, and others, "On-Demand Developer Documentation," *Software Maintenance and Evolution (ICSME)*, 2017. [Online]. Available: <http://www.inf.usi.ch/lanza/Downloads/Robi2017a.pdf>
- [6] J. Stylos and B. A. Myers, "The Implications of Method Placement on API Learnability," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: ACM, 2008, pp. 105–112. [Online]. Available: <http://doi.acm.org/10.1145/1453101.1453117>
- [7] J. Stylos, A. Faulring, Z. Yang, and B. A. Myers, "Improving API documentation using API usage information," in *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Sep. 2009, pp. 119–126.
- [8] J. Stylos and B. A. Myers, "Mica: A Web-Search Tool for Finding API Components and Examples," in *Visual Languages and Human-Centric Computing (VL/HCC'06)*, Sep. 2006, pp. 195–202.
- [9] E. L. Glassman, T. Zhang, B. Hartmann, M. Kim, and U. Berkeley, "Visualizing API Usage Examples at Scale," p. 12, 2018.
- [10] S. Jiang, A. Armaly, C. McMillan, Q. Zhi, and R. Metoyer, "Docio: Documenting API Input/Output Examples," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, May 2017, pp. 364–367.
- [11] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200. [Online]. Available: <http://doi.acm.org/10.1145/1065010.1065034>

Towards Scaffolding Complex Exploratory Data Science Programming Practices

Mary Beth Kery
 Carnegie Mellon University
 Pittsburgh, PA 15213
 mkery@cs.cmu.edu

I. INTRODUCTION

Although a wide range of professional and end-user programmers want to engage today with data science programming, this form of programming presents unique challenges. For instance, data science tasks typically require exploratory iterations: coding and running many different approaches to reach a desired result [1]–[3]. In a body of research building towards my thesis, I have interleaved behavioral studies of data scientists with systems building research towards scaffolding new forms of support for keeping track of iterations during this experiment-driven form of work.

II. DESIGNING VERSION CONTROL FOR EXPERIMENTATION

Our initial studies with data scientists identified several general programming practice barriers that individuals face, including tracking changes. Although tracking their experimentation was reported as a barrier, 48% of survey participants and 7 out of the 10 interview participants chose *not* to use well-established version control tools like Git in this context, even if they actively used Git in other projects [2]. In interviews, participants described using informal versioning techniques, like saving multiple copies of an analysis script to checkpoint it, or using code comments or otherwise unused “dead code” to keep multiple versions of a code snippet. These kinds of informal versioning techniques cause messy code structure, and thus are generally considered bad practice. Yet for experimentation, these behaviors were surprisingly common, with 4 of the 10 interview participants and 79% of survey participants keeping old code through commenting alone. We hypothesized this indicates real user needs unaddressed by current tooling: A) data scientists want to easily return to prior code in case a current exploration does not work out, B) data scientists want alternatives of their code easily viewable to compare. Just like a traditional scientist works to compare the effect of a number of interventions, data scientists are often not engineering towards a fixed specification so much as trying to understand a space of possible data manipulations.

Our first exploration of this design space stuck close to the above observed behavior. In Variolite, a user could draw a box around a snippet of code they would like to vary, as an alternative to commenting or duplicating it [2]. Within that

box, much like using a web browser, the user can switch tabs on the box in order to switch in-place what version of code they were using. Whichever code version was shown, was the one that would be executed at runtime (Figure 1).



```
v0 v0-1
def matchString(s):
    return difflib.SequenceMatcher(None, s, 'hello world').ratio()
```

Figure 1. Variolite variant box that contains two variants of a simple function

In a usability study and many informal subsequent discussions with data scientists, Variolite was positively received, allowing us to validate needs that this design was at least close to what data scientists wanted for their experimentation. However, this design is limited in two major ways. First, the underlying variational model of storing alternatives for only select snippets of code is inflexible for iteration. As the programmer works, their target for where in the code they are interested in evolves, which may quickly devolve into a convoluted clutter of these variant boxes and mixtures of snippets of code with rich history and snippets without in the same box. Second, although Variolite visualized which program output went with which code variants in a simple list, this list quickly becomes long, repetitive, and thus difficult to navigate [4], [5] as the user continuously edits and runs their code. We sought to more directly visualize how code variants affect code output to scaffold experimental questions like “Which data did I run to achieve this plot?”, “What model parameters have I tried so far and what was their effect?”, “Under what assumptions did I get this result?”, etc. Taking in account real data science tasks, we expanded what needs to be captured in version control in this domain from just code to be *artifacts*. An artifact is anything used in a data scientist’s code experimentation that might be needed to comprehend what a version *means*, including input, output, code, plots, notes etc.

To investigate this direction, we next conducted a study of computational notebooks, which are a form of code editing environment that have become popular among data scientists [6]. Computational notebooks allow the user to write and run code, output, notes, and other multimedia artifacts in the same document. In this study, we interviewed 21 professional data scientist users of Jupyter notebooks to understand how they iterated and experimented in computational notebooks [3]. Although this yielded many findings specific to computational

notebooks, many points about how users were handing version control through informal tactics were consistent with our prior studies. We also surveyed 45 data scientists, asking them to brainstorm how they would want to be able to retrieve a past version of their work, if they had a “magical oracle” that could always provide them with any version [3]. We used these results to begin to probe the hypothesis that data scientists understand their experimental versions in terms of the context of their artifacts. Indeed, many of the 125 magical queries participants generated involved getting back to an old version by referring to a particular output, the visual aspects of a plot, dataset used, and others, in addition to code and timestamp.

In our next iteration on Variolite, called Verdant, we prototyped an extension to Jupyter notebooks, in order to take advantage of what computational notebooks already do well: collecting relevant experimental artifacts in one place. Here, we sought to eliminate the limitations of Variolite’s versioning model, and allow users to ask version questions of all relevant artifacts in the Jupyter notebook. To be rid of the clutter nightmare of variant boxes that Variolite can cause, in this model we developed a succinct form of abstract syntax tree (AST) versioning such that every semantically meaningful piece of the program structure carries its own version history. Each time the user runs or saves their code, the model checks for any changed nodes of the AST, and if that node has been updated by edits, a new version of that node is recorded. In practice this means that if a user has 435 different versions of their entire document, but those contain only 3 unique values for a variable, the user can easily retrieve just those 3 unique variable values, which the variable has stored in its own history. Although counterintuitively this approach generates far more variational structure than Variolite, we lift the burden on the user for manually maintaining versions by elevating “versioning” to a first-class structure of the program. With this in place, the history is much more easily content addressable. Rather than scanning a long list of runs, by simply clicking on a code snippet or a plot, or a markdown note, the data scientist can find and view the history of that specific artifact. Figure 2 below shows how Verdant collects relationships among artifact versions to more directly answer questions, such as how to reproduce an output.

III. DISCUSSION & FUTURE WORK

My current focus is to take our initial prototype of Verdant, which contains design hypotheses of how users could ask experiment-oriented versioning questions, and use user-centered design to iterate. I aim to run a formal experiment with this tool, taking a version-finding scenario similar to [4], to test how different features of history visualization can allow a data scientists to *quickly* reach a prior version based on how they cognitively recall it. Following this work, I would like to use the experiment recording abilities of these designs to engage in behavioral experiments of how data scientists experiment in a more narrow problem domain, such as building a machine learning (ML) classifier. By extending

existing work such as [1] to research strategies and pitfalls data scientists face in ML development, I next aim to develop new editor tools to scaffold support for users to, given their experiment history so far, decide what approaches to try next.

The screenshot shows a software interface for 'Verdant'. At the top, there is a button labeled 'X' and a link 'to reproduce output:'. Below this is a table with columns 'IncidentNum', 'Category', 'Description', and 'Date'. Two rows are visible: one for incident 150060275 categorized as 'NON-CRIMINAL' with 'LOST PROPERTY' description and date 'Mo'; another for incident 150098210 categorized as 'ROBBERY' with 'ROBBERY, BODILY FORCE' description and date 'Su'. Below the table is a code editor window divided into two sections. The top section, labeled 'step #0 v0 Today 7:15 PM', contains the following Python code:

```
import sys,os
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

The bottom section, labeled 'step #1 v9 Today 7:15 PM', contains:

```
d_crime=pd.read_csv('./Police_Incidents.csv')
print d_crime.shape
```

Figure 2. A visualization in Verdant. By clicking on a version output in the editor, the user can see a recipe of code to run to reproduce that output.

ACKNOWLEDGEMENTS

This research was supported in part by a grant from Bloomberg L.P.. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect those of the funders.

REFERENCES

- [1] K. Patel, J. Fogarty, J. A. Landay, and B. Harrison, “Investigating statistical machine learning as a tool for software development,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2008, pp. 667–676.
- [2] M. B. Kery, A. Horvath, and B. A. Myers, “Variolite: Supporting Exploratory Programming by Data Scientists,” in *CHI*, 2017, pp. 1265–1276.
- [3] M. B. Kery, M. Radensky, M. Arya, B. E. John, and B. A. Myers, “The Story in the Notebook,” in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18*, 2018.
- [4] S. Srinivasa Ragavan, S. K. Kuttal, C. Hill, A. Sarma, D. Piorkowski, and M. Burnett, “Foraging Among an Overabundance of Similar Variants,” in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, San Jose, California, USA, 2016, pp. 3509–3521.
- [5] M. Codoban, S. S. Ragavan, D. Dig, and B. Bailey, “Software history under the lens: A study on why and how developers examine it,” in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015.
- [6] F. Pérez and B. E. Granger, “IPython: a System for Interactive Scientific Computing,” *Computing in Science and Engineering*, vol. 9, no. 3, pp. 21–29, May 2007.
- [7] M. B. Kery and B. A. Myers, “Interactions for Untangling Messy History in a Computational Notebook” in *Proceedings of VL/HCC*, 2018

Towards Supporting Knowledge Transfer of Programming Languages

Nischal Shrestha

Department of Computer Science
North Carolina State University
Raleigh, NC, USA
nshrest@ncsu.edu

I. INTRODUCTION

Today, there are hundreds of programming languages that are widely used. Programmers at all levels are expected to become proficient in multiple languages. Experienced programmers who have knowledge of at least one language are able to learn a second language much quicker than novices. However, the transfer process can still be difficult when there exists numerous differences from their previous language. Documentation, online courses and tutorials tend to present information geared towards novices. This type of presentation might suffice for beginners, but it doesn't support learning for experienced programmers [1] who would benefit from leveraging their knowledge of previous programming languages.

In my work, I explore teaching programming languages through the lens of *learning transfer*, which occurs when learning in one context either enhances (positive transfer) or undermines (negative transfer) a related performance in another context [2]. To investigate this approach, I created and evaluated a research tool called Transfer Tutor that teaches programmers R in terms of Python and Pandas, a data analysis library (see Fig. 1). The following design choices were made to explore learning transfer, applied to the topic of data frame manipulation: 1) highlighting similarities between syntax elements to support learning transfer 2) explicit tutoring on potential misconceptions 3) stepping through and highlighting elements of the snippets incrementally.

There are few studies examining transfer in the context of programming languages. Transfer of declarative knowledge between programming languages has been studied by Harvey and Anderson [3], which showed strong effects of transfer between Lisp and Prolog. Scholtz and Wiedenbeck [4] found that programmers suffer from negative transfer of Pascal or C knowledge when implementing code in a new programming language called Icon. Wu and Anderson [5] found problem-solving transfer for programmers writing solutions in Lisp, Pascal and Prolog which could improve programmer productivity. However, none of these studies investigated tool support.

Bower [6] explored a new teaching approach called Continual And Explicit Comparison (CAEC) to teach Java to students who knew C++. They found that students benefited from the continual comparison of C++ concepts to Java. Transfer Tutor

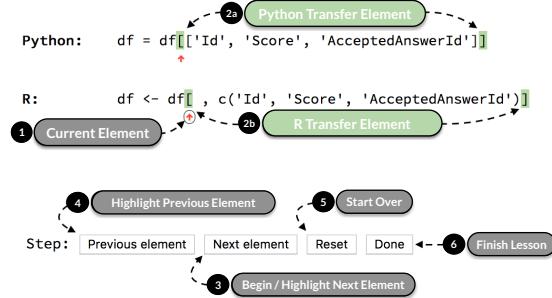


Fig. 1: The red arrow (1) indicates the currently highlighted syntax element for the code snippets in Python (2a) and R (2b). The stepper buttons are used to start the lesson or step forwards (3) and backwards (4) through the relevant syntax elements of the code snippets, reset (5) or end the lesson (6).

uses a similar teaching approach but provides interactivity, allowing programmers to visualize the differences between two languages with the use of highlights, which serves as affordances for transfer [7]. Further, Transfer Tutor allows programmers to step through the code, which helps them make a mindful abstraction of the concepts [8].

Fix and Wiedenbeck [9] developed and evaluated a tool called ADAPT that teaches Ada to programmers who know Pascal and C. Their tool helps programmers avoid implementation plans which contain negative transfers from Pascal and C, but is targeted primarily at the planning level. Transfer Tutor explicitly emphasizes both syntax and semantic issues by highlighting differences between the syntax elements in the code snippets of the two languages. Unlike ADAPT, Transfer Tutor is focused on transferring declarative knowledge [3], such as syntax rules, rather than procedural knowledge, such as implementation planning.

II. PRELIMINARY FINDINGS

I conducted a user study of Transfer Tutor with 20 participants from a graduate Computer Science course at North Carolina State University. A qualitative analysis on think-aloud protocols revealed that participants made use of learning transfer even without explicit guidance. The responses to a user satisfaction survey revealed additional insights on the design implications of future tools supporting transfer:

A. Affordances for supporting learning transfer

The majority of the participants found that incrementally stepping through the syntax elements was a useful feature as it helped them focus on one syntax element at a time and catch misconceptions on the spot. However, it prevented more advanced programmers from easily skipping explanations from the tool. Despite the usefulness of always-on visualizations in programming environments [10], allowing the programmer to activate explanations on-demand might be beneficial, using a mouse hover for example.

Reducing information load and allowing live code execution were two improvements suggested by the participants. This suggests Transfer Tutor needs to reduce information overload and balance the volume of explanation against the amount of code to be explained. One solution is to externalize additional explanations to documentation outside of the tool, such as web resources. Breaking up lessons into smaller segments could also reduce the amount of reading required. Future iterations of Transfer Tutor could include code execution, adapting explanations for the programmer's code.

B. Expert learning can benefit from learning transfer

The *expertise reversal effect* suggests that instructional techniques that are effective for novices can have negative consequences for experienced learners [11]. I have tried mitigating this effect by presenting explanations in terms of language transfer—in the context of a language that the programmer is already an expert in. Transfer Tutor serves as an instructional intervention: experienced programmers can use the tool to familiarize themselves with the language and over time reduce and eventually eliminate use of the tool.

III. FUTURE WORK

A. Automatic code translation and annotation

Transfer Tutor lacks support for easily finding the mappings between the syntax of two programming languages. The lesson designer needs to manually translate code from one language to another which is a tedious and error-prone process. Automatic code translation of code would help ease this process. For example, SMOP (Small Matlab and Octave to Python compiler)¹ is a transpiler that converts Matlab/Octave code to Python which is useful for code reuse. However, the resulting Python code is not useful for learning purposes as the programmer still needs to relate Python back to Matlab or Octave. The tool would be more useful if the generated code also contained annotations of the translation that took place so that programmers can better understand Python in relation to Matlab/Octave.

B. Mining for transfer issues

Researchers and instructors may find it difficult to identify the most important transfer issues. This could be solved by mining Q&A sites like Stack Overflow (SO). Using Truede's methodology [12], I performed a preliminary analysis on SO

posts tagged with both `<R>` and `<Pandas>`. I discovered that most programmers ask about how to translate a piece of code in Python/Pandas to R or vice-versa. Most accepted answers to these questions provide not only the equivalent piece of code in the target language, but also rich explanations that describe exceptions and gotchas. Mining could be a useful approach for collecting empirical data on transfer issues across programming languages.

C. Automatic design of transfer lessons

Currently, there is no easy method to create new transfer lessons for programming languages. Future tools could allow an instructor to specify the source and target language to generate a series of lessons automatically. If teaching Python, for example, the tool would first cover fundamental topics like variable assignment or for loops then slowly scale up to more advanced topics like list comprehensions. It would also be helpful if the tool generated interactive code examples and quizzes for each lesson to help learners test their knowledge. This addresses the limitations of Transfer Tutor regarding the lack of hands-on experience and the manual labor required to design a series of lessons. The two previous approaches—mining and automatic translation—serve as supplemental methods for this research direction.

IV. ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Chris Parnin, for his advice and support of this work. This material is based in part upon work supported by the National Science Foundation under Grant Nos. 1559593 and 1755762.

REFERENCES

- [1] L. M. Berlin, “Beyond program understanding: A look at programming expertise in industry,” *Empirical Studies of Programmers (ESP)*, vol. 93, no. 744, pp. 6–25, 1993.
- [2] D. N. Perkins, G. Salomon, and P. Press, “Transfer of learning,” in *International Encyclopedia of Education*. Pergamon Press, 1992.
- [3] L. Harvey and J. Anderson, “Transfer of declarative knowledge in complex information-processing domains,” *Human-Computer Interaction*, vol. 11, no. 1, pp. 69–96, 1996.
- [4] J. Scholtz and S. Wiedenbeck, “Learning second and subsequent programming languages: A problem of transfer,” *International Journal of Human-Computer Interaction*, vol. 2, no. 1, pp. 51–72, 1990.
- [5] Q. Wu and J. R. Anderson, “Problem-solving transfer among programming languages,” Carnegie Mellon University. Tech. Rep., 1990.
- [6] M. Bower and A. McIver, “Continual and explicit comparison to promote proactive facilitation during second computer language learning,” in *Innovation and Technology in Computer Science Education (ITiCSE)*, 2011, pp. 218–222.
- [7] J. G. Greeno, J. L. Moore, and D. R. Smith, “Transfer of situated learning,” in *Transfer on trial: Intelligence, cognition, and instruction*. Westport, CT, US: Ablex Publishing, 1993, pp. 99–167.
- [8] D. H. Schunk, “Learning theories,” *Prentice Hall Inc., New Jersey*, pp. 1–576, 1996.
- [9] V. Fix and S. Wiedenbeck, “An intelligent tool to aid students in learning second and subsequent programming languages,” *Computers & Education*, vol. 27, no. 2, pp. 71 – 83, 1996.
- [10] H. Kang and P. J. Guo, “Omnicode: A novice-oriented live programming environment with always-on run-time value visualizations,” in *User Interface Software and Technology (UIST)*, 2017, pp. 737–745.
- [11] S. Kalyuga, P. Ayres, P. Chandler, and J. Sweller, “The expertise reversal effect,” *Educational Psychologist*, vol. 38, no. 1, pp. 23–31, 2003.
- [12] C. Treude, O. Barzilay, and M.-A. Storey, “How do programmers ask and answer questions on the web?: Nier track,” in *2011 33rd International Conference on Software Engineering (ICSE)*, May 2011, pp. 804–807.

¹<https://github.com/victorlei/smop>

Creating Interactive User Interfaces by Demonstration using Crowdsourcing

Rebecca Krosnick

Computer Science & Engineering | University of Michigan, Ann Arbor

rkros@umich.edu

I. INTRODUCTION

People are becoming increasingly interested in creating their own digital content and media. This is evident in the enormous number of blogs, personal websites, and portfolios available online. Website templates and creation/hosting services (e.g., Wix, WordPress, Google Sites) have made it possible for even non-programmers to create websites. However, with these services, non-programmers are limited to templates or basic user interface elements and behaviors, lacking the ability to create truly custom web pages that satisfy their needs. More complex and custom user interfaces like digital games and software are virtually impossible for non-programmers to create; even visual programming (e.g., Blockly, GameMaker Studio 2) and data flow languages that try to make computing more approachable still require an understanding of programming and computing concepts. As simple as it is for the average person to sketch a User Interface (UI) on paper or describe it in words, I believe it should be just as easy for them to create the actual digital UI with all of the desired behaviors. Programming should not be a barrier to creating new things and sharing them with the world.

Programming by Demonstration (PbD) has been an approach previously explored to enable end-user programmers to create programs without writing program code. End-users instead demonstrate how their program should work for example scenarios. There has been a rich body of work in PbD, with a number of papers focused on building interactive UIs and games [1] [2] [3]. Although these systems proved promising in lab studies, PbD has not seen much adoption in commercial products, a couple reasons being: 1) often many demonstrations are needed for the PbD system to correctly infer the end-user's intended behaviors, and 2) it can be difficult for end-users to understand what exact demonstrations are needed [4].

In my future work I plan to address these challenges in an effort to make PbD a more feasible approach for enabling end-user programmers to build custom, interactive UIs.

#1: Make it more feasible to gather many demonstrations: In prior PbD systems, it was assumed that the single end-user would create all demonstrations and answer the system's clarifying questions. This requires much effort from one person, and can make using such a system undesirable. I propose applying crowdsourcing to this problem, to spread the effort across multiple people. I am currently designing a

crowdsourcing pipeline that leverages crowd workers to create PbD demonstrations capable of accurately satisfying the end-user's UI behavior requirements.

#2: Make it easier to gather the right demonstrations: To make it easier to gather demonstrations needed to correctly disambiguate and refine UI behaviors, I propose finding ways to guide crowd workers to create these demonstrations. I believe that by asking workers questions about what UI elements and properties a behavior is dependent on, the system can understand what new demonstration start-states or triggers would be informative and can ask workers to demonstrate the corresponding responses.

II. PBD FOR CREATING DYNAMIC UIs

As one step in the direction of enabling end-user programmers to build custom UIs, I have recently created Espresso [5], a PbD tool for building UIs with custom responsive behaviors, something that current template and website creation services do not support. Espresso does not require the user to write program code. With Espresso, a user starts with a static layout web page and then creates *keyframes* — examples of how the web page should look for different viewport widths — by directly manipulating UI elements in a WYSIWYG editor. Espresso can use a small set of keyframes to determine page layout for any viewport width. By default, the layout for a viewport width between two provided keyframes is the linear interpolation of the two keyframes' element property values, or a *smooth* transition. Espresso also supports discontinuous changes in layout as the viewport width is changed, for example a UI element being horizontally centered for small viewport widths and right-aligned for large viewport widths, which is enabled by the ability to set a *jump* transition between two keyframes. In a study I ran with participants who had minimal Cascading Style Sheets (CSS) experience [5], I saw that participants were able to build realistic responsive behaviors using Espresso. Although Espresso is effective for creating responsive UIs, it does not support creating more complicated behaviors, such as those in a digital game that may depend on interaction events and the state of various UI elements. Many more demonstrations would be needed to successfully encode such complicated behaviors using PbD. Using crowdsourcing could make creating a large number of demonstrations more manageable.

III. RELATED WORK IN CROWDSOURCING

Crowdsourcing is the act of making an open call for people to complete work. It is often used to scale human computation, which integrates human intelligence into a computational process to complete work better than either humans or machines could alone. Recent work has shown that continuous real-time crowdsourcing [6] can be used for building and powering UI prototypes based on end-user requests. Apparition [7], [8] enables an end-user to use natural language and hand-sketches to communicate UI requirements. The crowd then implements a higher-fidelity prototype matching the requirements, and can Wizard-of-Oz animation requirements. SketchExpress [9] builds on Apparition by enabling workers to create, save, and reuse animation behaviors, which the end-user can replay later. However, neither of these systems support creating truly automated interactive UIs. At run time, a human must either manually animate behaviors (in Apparition) or manually press “play” buttons to replay recorded behaviors (in SketchExpress). Behaviors dependent on state changes or user interaction events are not supported. By instead leveraging crowd workers to create PbD demonstrations, a system can infer a UI behavior model that can be applied to automatically render UI updates based on events and user interactions.

IV. FUTURE WORK

I am starting to design the crowdsourcing pipeline that will generate the PbD demonstrations necessary to define an end-user’s requested UI. An end-user requester will first describe their UI behavior requirements by text or audio. Each description will then be sent to a crowd worker, who will be asked to create relevant demonstrations. Like some prior PbD systems, we will likely ask the worker to demonstrate a “UI before-state” and events (e.g., user interaction, timer event, UI change event), and then the resulting “UI after-state”. A worker will likely need to provide multiple demonstrations for the UI to correctly exhibit the behavior they were assigned.

As with most crowdsourcing systems, the system should not blindly assume that any particular worker demonstration is correct. However, it would defeat the purpose to have the end-user check the validity of each worker demonstration; in that case, the end-user could have just spent their time creating all the demonstrations themselves. To address accuracy concerns while requiring zero or minimal work from the end-user, I plan to gather redundant demonstrations for the same (“before-state”, event) pair from multiple workers. For a previously created demonstration, its (“before-state”, event) could be passed to other workers, who would then be asked to demonstrate the expected “after-state”. The system would then need some intelligent, and likely automated, scheme for aggregating the redundant demonstrations in order to generate the most accurate demonstration of the requested behavior as possible. Although asking for redundant demonstrations will increase the total amount of work required, I claim that the amount of work required for any single worker will still be less than an end-user providing demonstrations alone, as

the system will not require every worker to complete every (“before-state”, event) demonstration.

Since crowd workers will not be expert users of PbD or this system, it will be particularly important to make creating the right demonstrations easy. “Good” demonstrations are ones that are meaningfully diverse, demonstrating a wide range of the state-space and clarifying behavior differences for small state changes. Achieving such diversity will be helpful in building robust UIs that satisfy the end-user’s requirements. It is likely that a worker may create a few initial demonstrations but then notice that the UI still does not completely satisfy the requester’s behavior requirements. I hope to help workers create demonstrations for new, relevant (“before-state”, event) pairs that would help the inference engine refine UI behaviors correctly. To do this, I intend to take prior (“before-state”, event) pairs and perturb them in meaningful ways, in order to generate new (“before-state”, event) pairs whose full demonstrations, completed by workers, would prove informative to the inference engine. To perturb (“before-state”, event) pairs in meaningful ways, I plan to also ask workers questions about the semantics of the requested UI behaviors, for example whether an element’s end location depends on its start location, or whether an element’s color depends on another’s.

Some other interesting questions to explore will be: What kinds of UI behavior descriptions can workers reliably understand, and which ones can they not? What kind of training will workers need to effectively use this system? Compared to an end-user performing all demonstrations themselves, how much faster can this crowdsourcing pipeline be?

I am excited about applying crowdsourcing to PbD as I think it could make PbD more feasible for end-user programmers. In general I am excited for a future where hopefully any person, regardless of technical expertise, can create custom programs and UIs that contribute to the world.

REFERENCES

- [1] B. A. Myers, “Peridot: creating user interfaces by demonstration,” in *Watch what I do*. MIT Press, 1993, pp. 125–153.
- [2] R. G. McDaniel and B. A. Myers, “Getting more out of programming-by-demonstration,” in *Proc. of CHI*. ACM, 1999, pp. 442–449.
- [3] M. R. Frank, P. N. Sukaviriya, and J. D. Foley, “Inference bear: designing interactive interfaces through before and after snapshots,” in *Proc. of DIS*. ACM, 1995, pp. 167–175.
- [4] B. A. Myers and T. J. J. Li, “Teaching intelligent agents new tricks: Natural language instructions plus programming-by-demonstration for teaching tasks.” Human Computer Interaction Consortium (HCIC), 2018.
- [5] R. Krosnick, S. W. Lee, W. S. Lasecki, and S. Oney, “Expresso: Building responsive interfaces with keyframes,” in *Proc. of VL/HCC*. IEEE, 2018.
- [6] W. S. Lasecki, K. I. Murray, S. White, R. C. Miller, and J. P. Bigham, “Real-time crowd control of existing interfaces,” in *Proc. of UIST*. ACM, 2011, pp. 23–32.
- [7] W. S. Lasecki, J. Kim, N. Rafter, O. Sen, J. P. Bigham, and M. S. Bernstein, “Apparition: Crowdsourced user interfaces that come to life as you sketch them,” in *Proc. of CHI*. ACM, 2015, pp. 1925–1934.
- [8] S. W. Lee, R. Krosnick, B. Keelean, S. Vaidya, S. D. O’Keefe, S. Y. Park, and W. S. Lasecki, “Exploring real-time collaboration in crowd-powered systems through a ui design tool,” in *Proceedings of the ACM Conference on Computer-Supported Cooperative Work and Social Computing*. ACM, 2018.
- [9] S. W. Lee, Y. Zhang, I. Wong, Y. Y., S. O’Keefe, and W. Lasecki, “Sketchexpress: Remixing animations for more effective crowd-powered prototyping of interactive interfaces,” in *Proc. of UIST*. ACM, 2017.

Assisting the Development of Secure Mobile Apps with Natural Language Processing

Xueqing Liu

*Department of Computer Science
University of Illinois Urbana-Champaign
Urbana, IL, USA
xliu93@illinois.edu*

I. INTRODUCTION

With the rapid growth of mobile devices and mobile apps, mobile has surpassed desktop and now has the largest worldwide market share [1]. While such growth brings in more opportunities, it also poses new challenges in security. Among the challenges, user privacy protection has drawn tremendous attention in recent years, especially after the Facebook-Cambridge Analytica data scandal in April 2018 [2].

Android controls users private data resources with the permission mechanism, e.g., user's location and contact list. The user must grant a permission before the app can access that private resource. Prior work shows that compared with targeted malware attacks, a more prevalent problem in mobile security is the *over-privileged* problem of *benign apps*. That is, apps request more permissions (e.g., location, contact list) than what they need [3]. On the one hand, such over-privileged problem can be exploited by malicious third party, allowing various security attacks such as inter-app permission leakage [4]. On the other hand, the prevalence of the over-privileged apps causes the difficulty for users to determine the legitimacy of the app. Prior work shows that users often do not understand why apps request certain permissions, and such confusion can cause their security concerns toward benign apps [5]. While a part of the confusion comes from the over-privileged problem, another important reason is that average user are not familiar with the technical permission purposes, e.g., for a camera app, the purpose of using the camera permission is straightforward. However, the purpose of using the location permission for geotagging is less straightforward.

As a result, two important security tasks for a benign app are: (1) how to detect/prevent security vulnerabilities in the development process? (2) how can the app educate users to improve their decision-making on the legitimacy of the app? While most existing work focuses on answering question (1), question (1) has not considered the role of users in the security model. Because Android permission model highly relies on user decisions, it is important for apps to assist users to make informative decisions.

In this thesis, I propose to study question (2). In particular, the proposed approach uses *natural language processing* (NLP) and statistical analysis techniques. First, I conduct an

empirical study for measuring the behaviors for developers to explain their apps' permission purposes for supporting users' security decision making. In the empirical study, I find that several explanation behaviors indicate that developers need assistance in providing such explanations. As a result, I propose a recommender system that assist developers by providing candidate permission explanations. The recommender system mines candidate permission-explaining sentences from similar apps' descriptions. Next, in the development stage, developers often have questions regarding secure development, e.g., StackOverflow users have asked how to use the `shouldShowRequestedPermissionRationales` API [6]. For such security related questions, it is helpful to support developers by providing a tool for question answering, which I plan to explore in future work.

In recent years, several papers enhances mobile app security with NLP approaches, e.g., [7]; however, such NLP techniques are far from being perfect. First, there exist large rooms for improving the accuracy of these tools. Further more, the latest advancements in NLP techniques also provide great potential for solving new NLP problems in mobile security. My thesis will seek such potential to better assist the overall process of secure development.

II. PRELIMINARY STUDIES

Our past work [8], [9] studies assisting developers to explain Android permission usages to app users. [9] studies developers' permission explanation behaviors in Android runtime permission rationales; [8] proposes a recommender system for assisting developers to explain permission usages.

Since Android 6.0 introduces the new runtime permission model, apps have tried to bridge such gaps by displaying message dialogs which explain the purposes of using the permissions. But how much proportion of apps leverage such functionality to provide explanations? How interpretable are existing apps' permission-explanations? To answer these questions, we leverage NLP and statistical analysis techniques [9] to study five aspects the interpretability of runtime permission rationales. We find the following conclusions: (1) less than one fourth of apps provide at least one rationales; (2) more apps provide rationales for straightforward permission purposes than for non-trivial permission purposes; (3) there exist some incorrectly explained rationales; (4) a large proportion of

rationales are redundant. These findings imply that apps may need assistance in creating permission explanations.

Following the findings in [9] and a few additional surveys, we identify the general difficulty in explaining permission usages. As a result, we propose a recommender system [8] to help developers with writing and improving their permission explanations. Our recommender system, CLAP, suggests candidate explanation sentences from similar apps' by leveraging NLP techniques. Our large-scale evaluation shows that CLAP can suggest permission explaining sentences of high quality. On the one hand, the accuracy of such sentences is more than 80%, outperforming the state-of-the-art NLP technique [7]. On the other hand, such sentences are highly interpretable. The interpretability include three aspects, i.e., concise, diverse, and expressing substantial purposes.

III. RELATED WORK

In recent years, a few pieces of work leverage NLP techniques for mobile security, e.g., [7]. Existing work in this direction motivated our early ideas in the preliminary studies.

The WHYPER tool [7] detects whether a sentence explains a permission by comparing the sentence with the permission's semantic model. The semantic model of a permission comes from the API documents controlled by that permission. One limitation of WHYPER is that the app must have explained the permission in its app description, which is often not the case. When the app has not explained the permission, or when the explanation needs further improvements, our CLAP tool [8] can help the developers by suggesting permission explanations from similar apps. In addition, our statistical analysis paper [9] finds that more apps explain permissions in their runtime rationales than in the app descriptions.

RiskMon [10] discovers that users have different expectation for different permissions in the same app. Users expect frequent permissions more than infrequent permissions. Based on this conclusion, we approximate the user expectation with permission frequency [9]. Such approximation allows us to quantitatively measure the correlation between user expectation and the common explanation behaviors among apps.

IV. FUTURE WORK

I pose the following research questions to guide my future work for assisting the development of secure apps:

RQ1. During the app development stage, the developer frequently ask questions on development forums such as StackOverflow, a large number of such questions are related to mobile security, e.g., in the following post [6], the developer asks about the usage of the API `shouldShowRequestPermissionRationale`. How to better assist developers search the correct answers to their security related questions?

RQ2. What is the potential for advanced NLP techniques in helping developers with general secure development? For example, can we leverage generative models (e.g., RNN) for automatically generating secure meta-data (e.g., app description, code comments)?

RQ3. How effective are permission explanations in helping users understand the permission purposes? What are some good properties for a sentence to effectively warn and educate users?

RQ4. Can NLP techniques help apps with the vulnerability checking, i.e., question (1) in Section I? For example, can we explain the taint flow using natural language?

RQ5. So far we have focused on the pre-development stage and the development stage, what is the potential for NLP techniques in the *post-development* stage, e.g., understanding security bug reports and user reviews?

We have recently begun studying RQ1. By comparing traditional retrieval models with deep neural network models, we find the latter can perform more accurate retrieval results. We plan to continue the study in this direction with the goal of improving existing work's retrieval performances.

REFERENCES

- [1] "Mobile marketing statistics compilation," <http://www.smartsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/>, accessed: 2018-06-29.
- [2] "Facebook and Cambridge Analytica data breach," https://en.wikipedia.org/wiki/Facebook_and_Cambridge_Analytica_data_breach, accessed: 2018-06-29.
- [3] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2011, pp. 627–638.
- [4] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek, "Covert: Compositional analysis of android inter-app permission leakage." *IEEE Trans. Software Eng.*, 2015.
- [5] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Proceedings of the Symposium On Usable Privacy and Security*, 2012, pp. 3–14.
- [6] "Should show request permission rationale API," [https://developer.android.com/reference/android/support/v4/app/ActivityCompat#shouldShowRequestPermissionRationale\(android.app.Activity.java.lang.String\)](https://developer.android.com/reference/android/support/v4/app/ActivityCompat#shouldShowRequestPermissionRationale(android.app.Activity.java.lang.String)), 2018, accessed: 2018-07-27.
- [7] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "Whyper: Towards automating risk assessment of mobile applications," in *USENIX*, 2013, pp. 527–542.
- [8] X. Liu, Y. Leng, W. Yang, C. Zhai, and T. Xie, "Mining android app descriptions for permission requirements recommendation," in *Proceedings of the International Requirements Engineering Conference*, 2018.
- [9] X. Liu, Y. Leng, W. Yang, W. Wang, C. Zhai, and T. Xie, "A large-scale empirical study on android runtime-permission rationale messages," in *VL/HCC*, 2018.
- [10] Y. Jing, G.-J. Ahn, Z. Zhao, and H. Hu, "RiskMon: Continuous and automated risk assessment of mobile applications," in *Proceedings of the ACM Conference on Data and Application Security and Privacy*, 2014, pp. 99–110.

Using Electroencephalography (EEG) to Understand and Compare Students' Mental Effort as they Learn to Program Using Block-Based and Hybrid Programming Environments

Yerika Jimenez

Department of Computer & Information Science & Engineering

University of Florida

jimenyer@ufl.edu

I. INTRODUCTION

In recent years, the US has begun scaling up efforts to increase access to CS in K-12 classrooms and many teachers are turning to block-based programming environments to minimize the syntax and conceptual challenges students encounter in text-based languages. Block-based programming environments, such as Scratch and App Inventor, are currently being used by millions of students in and outside of classroom. We know that when novice programmers are learning to program in block-based programming environments, they need to understand the components of these environments, how to apply programming concepts, and how to create artifacts. However, we still do not know how are students' learning these components or what learning challenges they face that hinder their future participation in CS. In addition, the mental effort/cognitive workload students bear while learning programming constructs is still an open question. The goal of my dissertation research is to leverage advances in Electroencephalography (EEG) research to explore how students learn CS concepts, write programs, and complete programming tasks in block-based and hybrid programming environments and understand the relationship between cognitive load and their learning.

II. BACKGROUND

Cognitive load refers to the amount of mental effort that is exerted by a student while performing a task or activity [1]. Sweller et al [2], identified three types of cognitive load. Intrinsic load is imposed by the inherent complexity of content, which relates to the extent to which various information elements interact. When the information interactivity is low, the content can be understood and learned one element at a time. In contrast, highly interactive information is more difficult to learn. While intrinsic load is generally thought to be immutable to instructional manipulation due to the inherent complexity of content [2], learning difficulty can be manipulated by controlling the contribution of germane and extraneous load. Germane load is mediated by the student's prior knowledge of domain,

cognitive and metacognitive skills. Therefore, it depends on the individual difference and learning characteristic of each student. Students experience germane load, when learning activity and/or material encourage high-order of thinking and challenge the learner at an appropriate level [3]. Extraneous load is the unnecessary mental burden that is caused by cognitively inappropriate design and presentation of information.

CS Education researchers have used different techniques to access and understand students programming knowledge such as content analysis of artifacts, and independent learning assessments [4]. These techniques provide researchers with valuable patterns of conceptual understanding and programming performance. But these techniques only provide a snapshot of the student and not the cognitive processes. Thus, it is difficult to understand the real-time challenges that students encounter when learning to program.

Research focused on students' cognitive processing of CS content currently uses data collected from self-reported cognitive load surveys and cognitive walkthroughs to understand how students' solve problems [4]. Morrison et al. researched students' perceived cognitive load or mental effort levels during two lectures using a self-reported cognitive survey [4]. They found that students perceived higher mental efforts while performing a task that required them to understand and use three CS concepts at the same time [4]. However, cognitive load surveys are a post hoc assessment of the cognitive load students experience and their self-reported nature means they are largely dependent on the reflective ability of the student. Measurement of cognitive load using cognitive walkthroughs of students during the learning task have been found to add to students' mental efforts as they are trying to explain their rationale and process [4]. Thus, these techniques by themselves are inadequate for understanding the factors that affect students' cognitive load. However, the use of neurophysiological devices such as EEG, can be used to capture real-time data about how students are engaging with content [5].

We propose the use of EEG to measure student's mental effort which can be further used to understand the students' cognitive activity and working memory load as they interact with the interface and perform programming tasks while learning CS in programming environments. Using EEG in conjunction with think-aloud techniques and learning assessments will allow us to understand students' mental effort challenges in real time.

III. EEG MENTAL EFFORT ANALYZER

EEG Mental Effort Analyzer (see Figure 1) was created because analyzing mental effort using neurophysiological data is difficult for researchers. Often time researchers would have an additional data source such as video data and need to combine or sync the two data sources together. EEG Mental Effort Analyzer is a tool that allows researchers of any field who are interested in understanding students' mental effort or cognitive workload as they learn and interact a new subject, an interface, and/or concept by using EEG and video as a data medium. EEG Mental Effort Analyzer allows researchers to analyze students mental effort or cognitive workload and identify specific times students experience high mental workload. The tool also allows researchers to take notes as they are analyzing videos of participants at particular timestamps. We are currently in the final phase of the development of this tool which will help us analyze the EEG and video data collected in our pilot study.

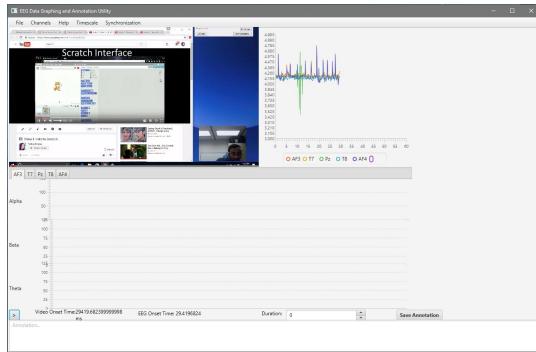


Figure 1: EEG Mental Effort Analyzer Interface

By early September, we hope to have analyzed our pilot study EEG data. This pilot study data was also analyzed using qualitative methods for which we will present a poster at VL/HCC 2018. Our pilot study was a two-part study designed to understand the role that usability of the programming environment has on novice participants while learning to program in Scratch and student's mental effort while coding in Scratch [6].

IV. FUTURE WORK

Due to the growing emphasis on integrating CS in middle school, the target population for this research is middle school students. During Fall 2018, I will conduct a study with my

target population. The research plan described in this section is composed of three phases. Phase 1 and Phase 2 will consist of same studies but with a different focus. Phase 1 will focus on block-based programming environment and Phase 2 on hybrid programming environment. Both phases will focus on identifying student learning and usability challenges and assess mental effort students' encounter when interacting with the interface, learning basic CS concepts and completing authentic activities/tasks in the programming environments.

In particular, I seek to identify specific aspects of learning activities and interactions that negatively impact student engagement. The goal of these studies is to understand how students are learning CS concepts and how they are interacting with block-based and a hybrid programming environment. In both phases, students will be asked to interact and perform nine tasks. Each of the tasks will have a different level of difficulty and will focus on a CS concept. While students are interacting and performing tasks, their EEG data will be recorded. The third phase will be a comparison between learning challenges and mental effort by students in block-based and hybrid programming environments. These studies will help us understand students' mental effort while learning to program in block-based and hybrid programming environment and what CS concepts students perceived as difficult.

Contributions of my dissertation research to the computer science education field: (1) A list of CS concepts through which students experience high levels of mental effort and reasons why they experience these; (2) Factors that contribute to students learning challenges in block-based programming environments; (3) A list of general design guidelines for all future block-based and hybrid programming environments taking into account students mental effort.

REFERENCES

- [1] J. Sweller, "Cognitive Load During Problem Solving: Effects on Learning", *Cognitive Science*, vol. 12, no. 2, pp. 257-285, 1988.
- [2] J. Sweller, J. van Merriënboer and F. Paas, *Educational Psychology Review*, vol. 10, no. 3, pp. 251-296, 1998.
- [3] L. Vygotsky, M. Cole, V. John-Steiner and E. Souberman, *Mind in Society*. Cambridge, Massachusetts: Harvard University Press., 1978.
- [4] B. Morrison, B. Dorn and M. Guzdial, "Measuring cognitive load in introductory CS", *Proceedings of the tenth annual conference on International computing education research - ICER '14*, 2014.
- [5] T. van Gog, F. Paas and J. Sweller, "Cognitive Load Theory: Advances in Research on Worked Examples, Animations, and Cognitive Load Measurement", *Educational Psychology Review*, vol. 22, no. 4, pp. 375-378, 2010.
- [6] Y. Jimenez, A. Kapoor and C. Gardner-McCune, "Usability Challenges that Novice Programmers Experience when Using Scratch for the First Time", *Processing Symposium on Visual Languages and Human-Centric Computing*, 2018.

The GenderMag Recorder's Assistant

Christopher Mendez, Andrew Anderson, Brijesh Bhuvan, Margaret Burnett

Oregon State University

Corvallis, Oregon, USA

{mendezc, anderan2, bhuvab, burnett}@eecs.oregonstate.edu

Abstract—Building software systems is hard work, with challenges ranging from technical issues to usability issues. If the technical issues are not addressed, the software cannot work -- but if the usability issues are not addressed, many potential users and customers are not even interested in whether it works. Further, usability must be inclusive: software needs to support diverse sorts of users. To help software professionals address gender-inclusive usability, we have created the GenderMag Recorder's Assistant tool. This Open Source tool is the first to semi-automate evaluating gender biases in software that is being designed, developed, or maintained. In this showpiece, we will demo the tool and encourage attendees to get involved in using it and improving upon it.

Keywords—GenderMag, gender inclusiveness

I. INTRODUCTION AND BACKGROUND

In this showpiece, we will demonstrate a new tool called the GenderMag Recorder's Assistant [7]. The tool is an Open Source project implemented as a Chrome extension, and is freely downloadable.

The Recorder's Assistant semi-automates use of the GenderMag method. GenderMag is a method to find gender bias “bugs” in software that is being designed, developed, or maintained [3]. GenderMag’s foundations lie in research on how people’s individual problem-solving strategies sometimes cluster by gender. At GenderMag method’s core are five problem-solving facets that matter to software’s gender-inclusiveness: a user’s motivations for using the software, their information processing style, their computer self-efficacy, their attitude towards risk, and their ways of learning new technology.

Evaluations of GenderMag’s validity and effectiveness have produced strong results. In a lab study, professional UX researchers were able to successfully apply GenderMag, and over 90% of the issues it revealed were validated by other empirical results or field observations, with 81% aligned with gender distributions of those data [3]. GenderMag was also used to evaluate a Digital Library interface, uncovering significant usability issues [4]. In a field study evaluating GenderMag in 2-to 3-hour sessions at several industrial sites [2, 5], software teams analyzed their own software using GenderMag, and found gender-inclusiveness issues in 25% of the features they evaluated. In Open Source Software (OSS) settings, OSS professionals used GenderMag to evaluate OSS tools and infrastructure and found gender-inclusiveness issues in 32% of the use-case steps they considered [6]. In a longitudinal study at Microsoft, variants of GenderMag were used to improve at least 12 teams’ products [1].

II. THE RECORDER'S ASSISTANT

The Recorder's Assistant is the first tool to semi-automate the identification of gender bias “bugs” in the user-facing layer of software. VL/HCC attendees who build or evaluate visual languages and interfaces can use it to evaluate the systems they are helping to design, develop, or maintain.

To use the Recorder's Assistant, a software team navigates via the browser to the app or mockup they want to evaluate, then starts the tool from the browser menu. The main sequence is to view a persona (Fig. 1(c)) and proceed through the scenario of their choice from the persona’s perspective, one action at a time. At each step, the tool’s “context-specific capture” captures screenshots about the action the team selects (Fig. 1(a)), and records the answers to questions about it (Fig. 1(b)). The tool saves this sequence of screenshots and questions/answers to form a gender-bias “bug report.”

The full VLHCC’18 paper [7] describes the tool and presents an empirical evaluation.

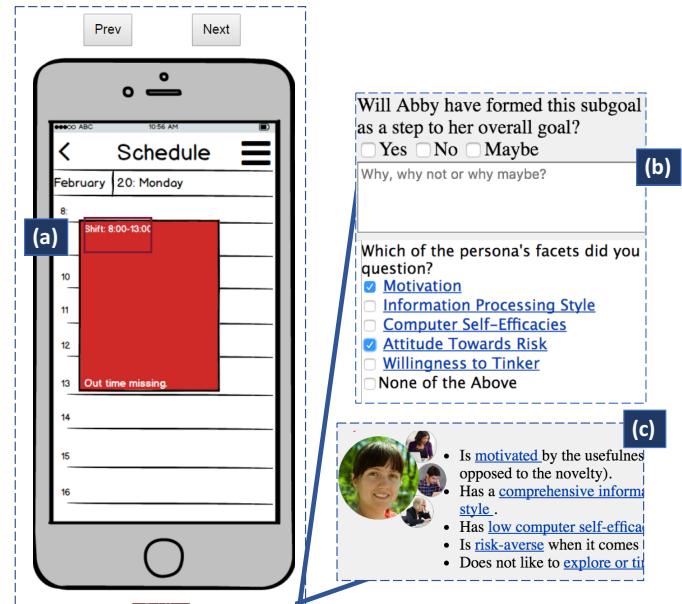


Fig. 1: The Recorder's Assistant tool during an evaluation of a mobile time-and-scheduling app. (Left): The app being evaluated is displayed with (a) a rectangle around the action the evaluators are deciding if a user like “Abby” will take. (Right): A blow-up of portions of the GenderMag features for the app: (b) the GenderMag question the team is answering at the moment, including a checklist of Abby’s facets; and (c) a summary of the persona the team has decided to use (in this case, Abby).

This work was supported in part by NSF 1314384 and 1528061.

III. HOW WE WILL PRESENT THE TOOL

We will present the tool during the Showpiece Reception via live demo's and a poster. A short video of a GenderMag session is also available at <http://gendermag.org/>.

IV. CONCLUDING REMARKS

The GenderMag Recorder's Assistant is an Open Source project. We invite people to download and/or contribute to it at <http://gendermag.org/>.

REFERENCES

- [1] M. Burnett, R. Counts, R. Lawrence, H. Hanson, Gender HCI and Microsoft: Highlights from a longitudinal study, IEEE VL/HCC, pp. 139-143, 2017.
- [2] M. Burnett, A. Peters, C. Hill, and N. Elarief, Finding gender inclusiveness software issues with GenderMag: A field investigation, ACM CHI, pp. 2586-2598, 2016.
- [3] M. Burnett, S. Stumpf, J. Macbeth, S. Makri, L. Beckwith, I. Kwan, A. Peters, and W. Jernigan, GenderMag: A method for evaluating software's gender inclusiveness. *Interacting with Computers* 28(6), pp. 760-787, 2016.
- [4] S. Cunningham, A. Hinze and D. Nichols, Supporting gender-neutral digital library creation: A case study using the GenderMag Toolkit. *Digital Libraries: Knowledge, Information, and Data in an Open Access Society*, pp. 45-50, 2016.
- [5] C. Hill, S. Ernst, A. Oleson, A. Horvath and M. Burnett, GenderMag experiences in the field: The whole, the parts, and the workload, IEEE VL/HCC, pp. 199-207, 2016.
- [6] C. Mendez, H. S. Padala, Z. Steine-Hanson, C. Hildebrand, A. Horvath, C. Hill, L. Simpson, N. Patil, A. Sarma, M. Burnett, Open Source barriers to entry, revisited: A sociotechnical perspective, ACM/IEEE ICSE, pp. 1004-1015, 2018.
- [7] C. Mendez, Z. Steine-Hanson, A. Oleson, A. Horvath, C. Hill, C. Hildebrand, A. Sarma, M. Burnett. Semi-automating (or not) a socio-technical method for socio-technical systems. IEEE VL/HCC 2018 (to appear).

Fritz: A Tool for Spreadsheet Quality Assurance

Patrick Koch

AAU Klagenfurt

Klagenfurt, Austria

Email: Patrick.Koch@aau.at

Konstantin Schekotihin

AAU Klagenfurt

Klagenfurt, Austria

Email: Konstantin.Schekotihin@aau.at

Abstract—While spreadsheets are widely used for business-related tasks, they are mostly handled by novice users instead of professional programmers. Consequently, those users often are not aware of quality issues in their spreadsheet programs that may lead to faults with significant adverse effects. In this work, we therefore present a tool, called FRITZ, to support users in checking and improving the quality of their spreadsheets. The tool enriches the traditional spreadsheet visualization scheme by including visual feedback about certain structural and quality aspects. This allows for easier cognition of a spreadsheet’s layout, and helps users to detect and comprehend irregularities within it. Furthermore, FRITZ highlights suspicious (smelly) cells, such as complex formula cells or empty input cells, that are prone to introduce errors. In contrast to other smell detection tools, FRITZ also warns against smells that point out structural irregularities.

Index Terms—Software tools, Spreadsheet programs, Software quality

I. INTRODUCTION

Due to rising popularity of end-user programming, most programs today are created by domain experts in need of computational power, instead of professional software developers [1]. Spreadsheets, in particular, provide vital computational capabilities for users in the public and private sectors alike, and are often used for financial modelling as well as management tasks. Faults in such spreadsheet models, however, can have serious consequences. For example, the Canadian power generation company TransAlta lost \$24 million (i.e. 10 % of TransAlta’s annual profit) due to a spreadsheet error [2]. Unfortunately, this incident is not an outlier, as emphasized by the list of recent spreadsheet debacles that is maintained by the European Spreadsheet Risk Interest Group¹.

Many of these debacles could likely have been prevented by the application of rigorous quality assurance practices for spreadsheets. However, while the scientific community successfully proposed numerous promising techniques to improve spreadsheet quality, the support for such techniques in common spreadsheet editors remains sparse. As part of our research efforts into spreadsheet quality aspects, we therefore developed FRITZ, a quality assurance tool for spreadsheets that incorporates previous and ongoing research.

II. FRITZ

FRITZ analyses and visualises spreadsheets. Figure 1 demonstrates the tool’s UI, showing a visually enhanced loan

calculation spreadsheet. The visualization of spreadsheets in FRITZ focuses on the following three key aspects:

First, FRITZ enriches the traditional spreadsheet UI by allowing users to highlight certain spreadsheet-specific attributes: e.g. a user can select to emphasize groups of related cells using distinct background colors, or to point out cells that are related to a specific selection using cell borders. This allows the user to visually identify, for example, the input and output cells of a spreadsheet, or cells that have formulas which differ from the formulas of neighboring cells. Highlighting also makes it easier to spot formula cells whose content has been accidentally overwritten by constant values.

Second, FRITZ provides warnings for inferred issues that affect either individual cells or groups of cells. These issues are encoded by spreadsheet smells: specific procedures that, like code smells, point out possible problems such as complex formulas, missing inputs, and problematic dependencies [3]–[5]. In addition to smells which are already known from literature, FRITZ also detects a set of novel, structural smells.

Third, FRITZ provides detailed contextual information for selected parts of a spreadsheet, e.g. information about the content, references, and detected smells of a selected cell. A separate UI window presents this info, organized using tabs, each informing about a specific aspect of the selection.

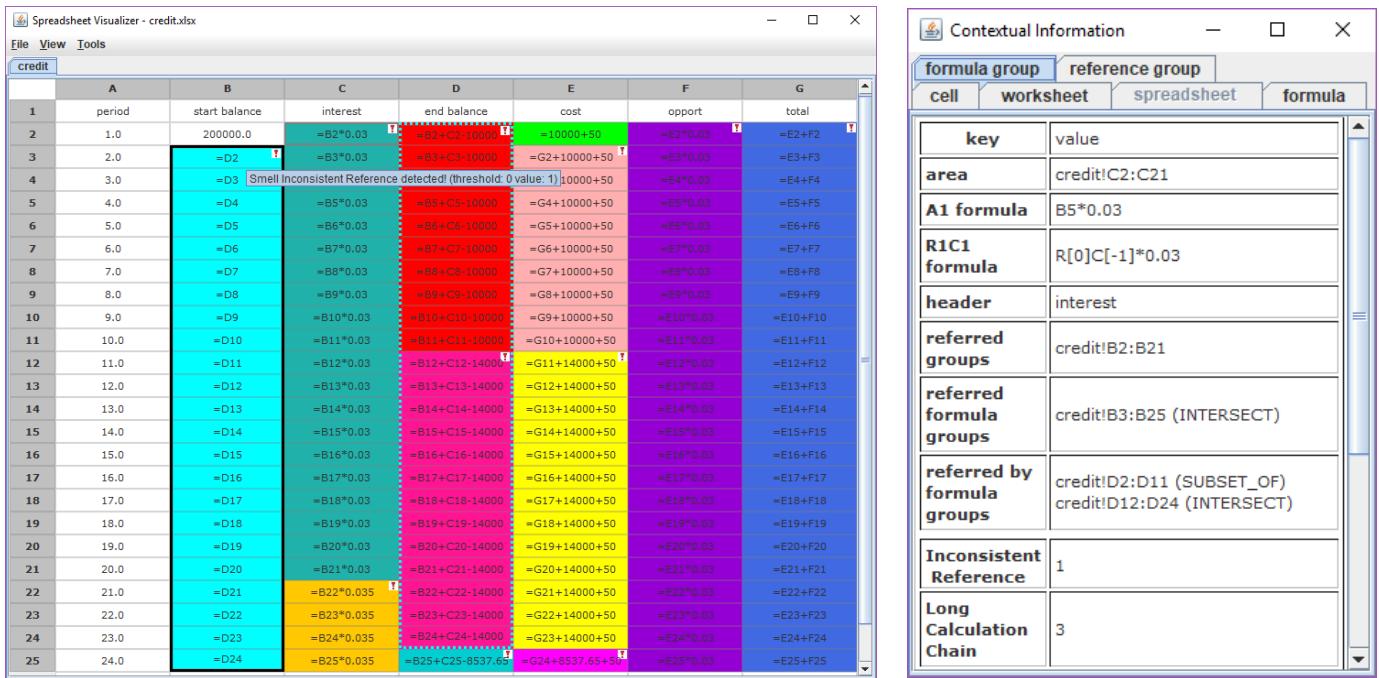
FRITZ is a research prototype² that resulted from our work on static spreadsheet analysis [6], where we focused on the automatic identification of header cells, cell groups, and computation blocks within spreadsheets. In ongoing but as of yet unpublished work, we used the inferred information to formulate structural smells for spreadsheet that are also detected by FRITZ. Additional features of the tool include the analysis and visualization of static aspects like named ranges, and the batch analysis of a collection of spreadsheets.

The intended audience of FRITZ are all people working with spreadsheets, e.g.: (1) a bank employee who analyses the spreadsheet his predecessor used to calculate a customer’s risk; (2) a company’s IT expert that was asked to fix an error in a given spreadsheet; or (3) a group of researchers who want to ensure that their results are free of errors before publication.

III. PRESENTATION

We will present the tool as a live demo to invite discussion about possible improvements and perspectives for future research. Moreover, a practical demonstration video of the tool is available at: <https://youtu.be/fk7vWcNHM40>.

¹Available at <http://www.eusprig.org/horror-stories.htm>.



(a) Example spreadsheet opened in FRITZ with active visualization of groups of formulas, reference-indicating mouse-over, and warning of group smells.

(b) Contextual information window, showing the formula group information of cell C5.

Fig. 1: Example spreadsheet, calculating the projected development of a consumer credit.

IV. RELATED WORK

FRITZ's static analysis approach is based on the ideas of Abraham and Erwig about automatic header inference [7]. Other approaches for structure inference and visualization in spreadsheets include the work of Koci *et al.* [8], who used machine learning to classify cells into different roles, and Hermans *et al.* [9], who proposed a tool for spreadsheet dataflow visualization. The works of Hermans *et al.* [3], [4], Cunha *et al.* [5], and Dou *et al.* [10] count among previous works to detect spreadsheet smells. However, to our knowledge, FRITZ is unique in its combination of structure inference as well as visualization, and combining this information with warnings about established and novel spreadsheet smells.

V. CONCLUSIONS & OUTLOOK

In this work, we presented FRITZ, a tool to support quality assurance efforts for spreadsheet users. The tool aids users in the understanding of concepts and relations that are hard to grasp using default spreadsheet representations. Utilizing this knowledge, users are better prepared to detect and fix faults in their spreadsheets, as well as to adapt and expand existing spreadsheet layouts. Moreover, raising the awareness of spreadsheet creators and users for the underlying structural aspects of the medium is likely to enhance the overall quality of spreadsheets in the future. In subsequent work, we plan to conduct a study to assess the usability of the tool, as well as to further extend the tool's functionalities by including the results of ongoing research into quality assurance techniques for spreadsheets.

ACKNOWLEDGMENT

The work described in this paper has been funded by the Austrian Science Fund (FWF) project *DEbugging Of Spreadsheet programs (DEOS)* under contract number I2144.

REFERENCES

- [1] A. J. Ko, R. Abraham, L. Beckwith, A. F. Blackwell, M. M. Burnett, M. Erwig, C. Scaffidi, J. Lawrence, H. Lieberman, B. A. Myers, M. B. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck, "The state of the art in end-user software engineering," *ACM Comput. Surv.*, vol. 43, no. 3, pp. 21:1–21:44, 2011.
- [2] T. G. Patrick Brethour and Mail, "Human error costs transalta \$24 million on contract bids," 2003, last visited: July, 13th 2018. [Online]. Available: <https://beta.theglobeandmail.com/report-on-business/human-error-costs-transalta-24-million-on-contract-bids/article18285651/>
- [3] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting and visualizing inter-worksheet smells in spreadsheets," in *ICSE*. IEEE Computer Society, 2012, pp. 441–451.
- [4] ———, "Detecting code smells in spreadsheet formulas," in *ICSM*. IEEE Computer Society, 2012, pp. 409–418.
- [5] J. Cunha, J. P. Fernandes, P. Martins, J. Mendes, and J. Saraiva, "Smellsheet detective: A tool for detecting bad smells in spreadsheets," in *VL/HCC*. IEEE, 2012, pp. 243–244.
- [6] P. W. Koch, B. Hofer, and F. Wotawa, "Static spreadsheet analysis," in *ISSRE Workshops*. IEEE Computer Society, 2016, pp. 167–174.
- [7] R. Abraham and M. Erwig, "Header and unit inference for spreadsheets through spatial analyses," in *VL/HCC*. IEEE Computer Society, 2004, pp. 165–172.
- [8] E. Koci, M. Thiele, O. Romero, and W. Lehner, "A machine learning approach for layout inference in spreadsheets," in *KDIR*. SciTePress, 2016, pp. 77–88.
- [9] F. Hermans, M. Pinzger, and A. van Deursen, "Breviz: Visualizing spreadsheets using dataflow diagrams," *CoRR*, vol. abs/1111.6895, 2011.
- [10] W. Dou, S. Cheung, and J. Wei, "Is spreadsheet ambiguity harmful? detecting and repairing spreadsheet smells due to ambiguous computation," in *ICSE*. ACM, 2014, pp. 848–858.

Code review tool for Visual Programming Languages

Giuliano Ragusa

FCT/UNL

Almada, Portugal

g.ragusa@campus.fct.unl.pt

Henrique Henriques

OutSystems SA

Linda-a-Velha, Portugal

henrique.henriques@outsystems.com

Abstract—Code review is a common practice in the software industry, in contexts spanning from open to close source, and from free to proprietary software. Modern code reviews are essentially conducted using cloud-based dedicated tools. Existing review tools focus in textual code. In contrast, support of low-code software languages, namely Visual Programming Languages (VPLs), is not readily available. This presents a challenge for the effectiveness of the review process with a VPL. This showpiece will present VPLreviewer, a code review tool for VPLs. VPLreviewer provides a wide range of mechanisms previously not available to a VPL. It is expected to improve communication among the stakeholders who have to review artifacts constructed with VPLs, with mechanisms that are easy to learn, use and understand.

I. INTRODUCTION

Code review's importance has already been proven. Several researchers provided evidence on traditional code inspection's benefits, especially in terms of defect finding [1], [2], [4]. And although finding bugs is important, the foremost reason for introducing code review in big companies, such as Google, is to improve code understandability and maintainability [3].

Visual Programming Languages (VPLs) have substantially increased their presence in the software industry in recent years. Yet, mechanisms to help increase software quality have not evolved accordingly. Thus, visual artifacts are not supported by existing textual programming languages code review tools, hampering the code review process.

OutSystems was used as a case study for VPLreviewer. The company produces a low-code platform that allows to visually develop entire applications. Even though at OutSystems code review is a concern, the lack of tools for VPLs makes reviewing them a major problem. The code review still happens and is encouraged. However, it isn't productive and has a negative impact on delivery speed.

II. THE TOOL

We developed a web-based tool to assist the code reviews of VPLs. The tool contains a wide range of features inspired in both already existent software for textual code review and problems identified in a visual programming context at OutSystems, such as: not being able to comment on a specific

change, not knowing the context or the order of the changes, etc.

VPLreviewer is a generic code review tool for VPLs. Instead of directly consuming the artifact from a specific visual language, we opted to feed the tool with a specific JSON structure containing all information of an artifact. Although the first plugin was developed for OutSystems, different plugins can be easily created to support other languages (e.g. Petri nets).

The solution's architecture encapsulates together two main components, making them work as one unique body from an outsider perspective. The architecture is split to allow the logic to be refactored or replaced without impacting the UI.

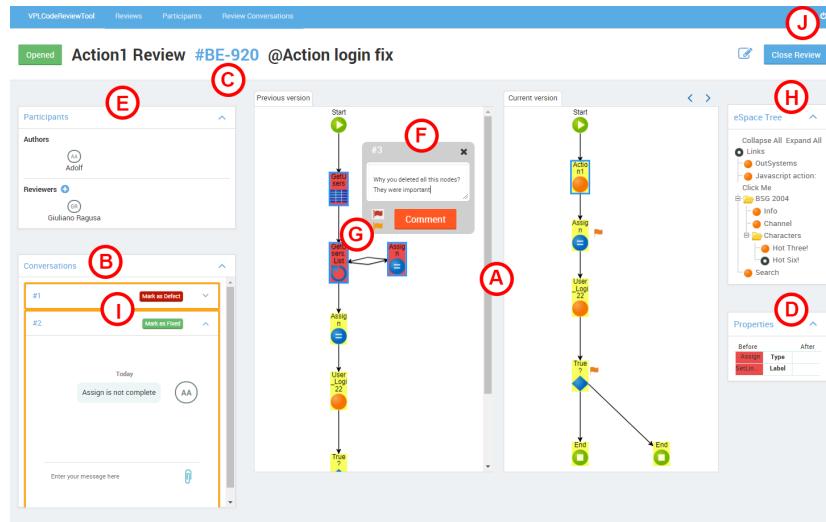
These components are:

- **Core services.** Represents the back-end of the tool. They are responsible for converting and storing the visual artifacts into tool-readable structures. This pre-processing enables the manipulation of each of the visual elements.
- **Tool.** The tool is able to compute differences between versions and present them side-by-side in a visual manner, along with all features designed to help participants conduct code reviews.

The user interface is the only component end users have access to. With its straightforward design and intuitive features, participants can focus their attention on detecting defects and consequently increasing code quality.

Figure 1 shows the tool's interface, while reviewing a visual artifact. All tool features were inspired by successful implementations of code review tools and from issues identified upon interviewing low-code developers at OutSystems. These include:

- **Diff-viewer** a side-by-side view (A) of two different versions of the same artifact;
- **Changes highlight** different color highlighting to changes for quickly understanding what changed. Red for deleted elements, yellow for modified and green for added;
- **Threads of discussion** participants can select a single or multiple nodes and create new discussion threads (B);
- **Issue Tracking Software integration** JIRA Atlassian integration for more extended context on change (C);
- **Properties** all changes have a properties table (D) to more in-depth context of a specific change;

Fig. 1. Tool's user interface (source at <https://goo.gl/i9VHd4>)

- **Files list** Tree listing of all files (H) related with the review;
- **Participants list** Ability of adding participants (E) as needed;
- **Notifications** Automatic notifications to every participant whenever a review is updated;
- **Data** stores data of each review and provides metrics such as number of defects found, average time per review, etc.

The tool presents a side-by-side view with all the differences highlighted. The reviewer will be able to open threads of conversations by simply right clicking (F) on any node and all threads opened will be listed on the side. The participant can chose the importance of his comment by switching the thread's flag (G) from warning (orange) to critical (red). By clicking any highlighted change, the reviewer will be able to get all the information on that specific change, giving him a more accurate context. If needed, the reviewer can click on the issue's hyperlink which will redirect him directly to the company's issue tracking platform. Reviews are only closed (J) when they need no more iterations and all critical threads of conversations are marked as fixed (I).

All in all, our main objective with this tool is to support, but not restrict the code review process, providing a flexible and lightweight tool. Our tool minimizes the effort devoted to administrative aspects such as scheduling meetings, encouraging attendance, and recording review comments. Using VPLreviewer makes it effortless to invite reviewers, distribute artifacts and gather feedback. Instead of recording issues on separate log forms, the tool lets reviewers insert their comments in context, right next to the visual code element in question, facilitating discussions among the reviewers on issues that are brought up.

III. IMPACT FOR THE VL/HCC COMMUNITY

VPLreviewer is of interest to the VL/HCC community because, even tough VPLs have been rapidly evolving, code

review software has stagnated on textual languages. Our work is an attempt to provide support for the code review process and increase the quality of the software developed with VPLs.

We plan to release VPLreviewer to the VPLs industry in the near future. Therefore, we would benefit from demonstrating the tool during the VL/HCC showpiece presentation session by allowing a community of experts to go through the tool and give us feedback to improve it before its release.

IV. PRESENTATION

The approach and tool presented by this showpiece paper will be further demonstrated using a screen-cast video (available at https://youtu.be/wgnZ_c235NQ). An author will explain the tool features and how the code review process is conducted. In addition to the video, attendees will also be able to test the tool with a variety of visual artifacts, in both author and reviewer perspective.

V. ACKNOWLEDGMENTS

The authors would like to thank OutSystems for all support given throughout this project.

REFERENCES

- [1] Fagan, Michael. "Design and code inspections to reduce errors in program development." *Software pioneers*. Springer, Berlin, Heidelberg, 2002. 575-607.
- [2] Laitenberger, Oliver, et al. "An experimental comparison of reading techniques for defect detection in UML design documents." *Journal of Systems and Software* 53.2 (2000): 183-204.
- [3] Sadowski, Caitlin, et al. "Modern code review: a case study at google." *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ACM, 2018.
- [4] Baccelli, Alberto, and Christian Bird. "Expectations, outcomes, and challenges of modern code review." *Proceedings of the 2013 international conference on software engineering*. IEEE Press, 2013.

Automated Test Generation Based on a Visual Language Applicational Model

Mariana Cabeda

FCT/UNL

Lisboa, Portugal

Email: m.cabeda@campus.fct.unl.pt

Pedro Santos

OutSystems

Linda-a-Velha, Portugal

Email: pedro.santos@outsystems.com

Abstract—This showpiece presents a tool that aids OutSystems developers in the task of generating test suites for their applications in an efficient and effective manner. The OutSystems language is a visual language graphically represented through a graph that this tool will traverse in order to generate test cases.

The tool is able to generate and present to the developer, in an automated manner, the various input combinations needed to reach maximum code coverage, offering a coverage evaluation according to a set of coverage criteria: node, branch, condition, modified condition-decision and multiple condition coverage.

Index Terms—Software test automation, Software test coverage, OutSystems language, Visual Programming Language, OutSystems applicational model.

I. DESCRIPTION

A. Introduction

The OutSystems [1] language, classified under Visual Programming Languages (VPLs), allows developers to create software visually by drawing interaction flows, UIs and the relationships between objects. Low-code tools reduce the complexity of software development bringing us to a world where a single developer can create rich and complex systems in an agile way, without the need to learn all the underlying technologies [2]. As OutSystems aims at rapid application development, automating the test case generation activity, based on their applicational model, along with coverage evaluation, will be of great value to developers using OutSystems.

Software testing is a quality control activity performed during the entire software development life-cycle and also during software maintenance [3]. Two testing approaches that can be taken are manual or automated. While for manual testing, the test cases are generated and executed manually by a human sitting in front of a computer carefully going through application screens, trying various usage and input combinations; in automated testing both the tasks of generation and execution of the test cases can be executed resorting to tools. The tool hereby presented covers the aspect of the test case generation and not its execution.

B. Tool

This showpiece introduces a tool that aims at generating, in an automated manner, test cases for applications developed in

the visual language OutSystems.

The algorithm behind this tool takes on the visual source code of an OutSystems application and generates all the necessary input combinations so that the set of generated test cases would be able to reach the entirety of its nodes and branches, or detect and identify unreachable execution paths, which in practice correspond to dead code.

As the OutSystems language is mainly visual and represented graphically through a graph, this tool resorts to graph search algorithms, breadth and depth-first search, in order to traverse these graphs and retain all necessary information.

Due to the extensibility of the OutSystems model, this tool currently supports an interesting set of nodes related to the logic behind client/server applications. Fig. 1 shows said nodes integrated within a simple graph example.

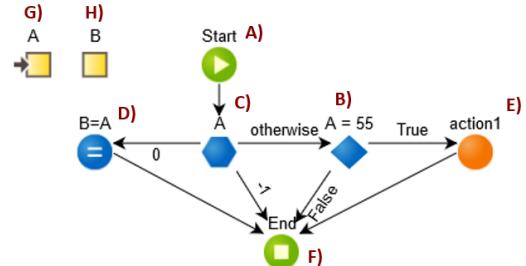


Fig. 1. OutSystems language nodes considered in this work: *Start* node (A) marks the start of a procedure; *If* node (B) expresses an if-then-else block behaviour; *Switch* node (C) representing a switch block behaviour; *Assign* node (D) indicating the attribution of values to variables; *Execute Action* node (E) represents a call to another procedure; *End* node (F) marks the end of a procedure. G) and H) represent input and local variables, respectively.

Along with the various input combinations that should be tested in order to achieve maximum code coverage and the identification of unreachable execution paths, it is also provided information on some warnings such as when variables are defined but never used in the trace of code analysed.

Software test coverage is a measure used to describe the degree to which the source code of a program is executed when a particular test suite runs. This tool evaluates both the overall test suite as well as subsets of it in terms of node, branch, condition, modified condition-decision and multiple condition coverage [4-6].

For this, the algorithm traverses the graph, from the Start node consecutively following the nodes outgoing branches,

applying a cause-effect graphing methodology [7] in order to reduce the generation of redundant combinations, and employing a boundary-value analysis [7,8] whenever a new decision point is reached in order to identify the values that should be tested for each individual condition.

The final test suite generated is prioritized according to two criteria: they are first organized in terms of the combined coverage they provide for both branches and nodes; the second criteria takes into account the number of decisions the path corresponding to this test case encounters. This prioritization is also complementary, meaning that when the first "best" test case is found, the second test case to be displayed is the one that, together with the first one, helps to cover more nodes and branches. The same goes for the third pick and so on. This means that the first x test cases presented are the ones that will cover the most nodes and branches and no other combination of x test cases will be able to cover more code.

Fig. 2 shows the prototype for this tool, where the set of test cases generated are displayed in (A), with some warnings identified in (B) and the coverage results in (C).

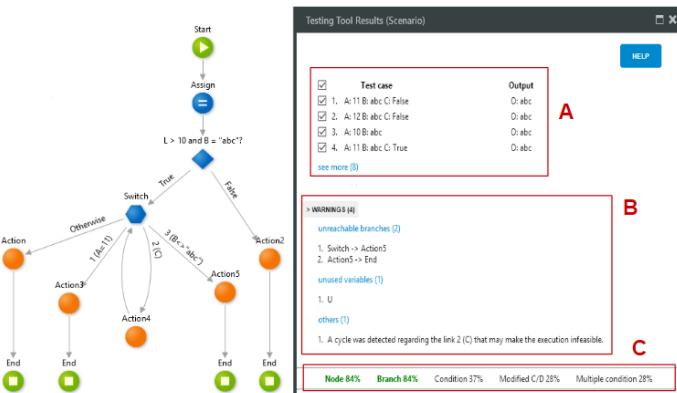


Fig. 2. Tool prototype window (expanded image at: <https://goo.gl/3yRWkA>)

A program is tested in order to add some value to it. This value comes in the form of quality and reliability, meaning that errors can be found and afterwards removed. This way, a program is tested, not only to show its behaviour but also to pinpoint and find as many errors as possible. Thus, there should be an initial assumption that the program may contain errors and then test it [8].

Nowadays, there is a high need for quick-paced delivery of features and software to customers, so automating tests is of the utmost importance. One of its several advantages is that it releases the software testers of the tedious task of repeating the same assignment over and over again, freeing up testers to other activities and allowing for a variety in the work as well as opening space for creativity. These factors are claimed to improve testers motivation at work [9].

As OutSystems aims at rapid application development, the automation of the test case generation activity, based on their applicational model, along with coverage evaluation, will be of great value to developers using OutSystems.

This tool is also of relevance to the VL/HCC community as it presents a solution for an issue that is very common within the visual languages paradigm. As the development of applications is still dominated by Textual Programming Languages (TPLs), a number of tools already allow automated testing over TPLs, but the same variety does not apply to VPLs. Tools such as the one here presented help increase the value brought by VPLs to developers.

II. PRESENTATION

This showpiece will be presented through video, showcasing its features (available at: <https://youtu.be/8GsY8NTNXdk>) as well as a demonstration that will involve the participation of users, consisting on an interactive exercise where the user will be able to experience the advantages brought on by this tool. This demonstration will consist of a simple two-part exercise, taking no longer than ten minutes, where one part will include the tool and the other will not. The results and feedback from this demonstration will be recorded for the purpose of evaluation of the tool.

Complementing this demonstration, there will also be a poster showcasing this tool's features alongside a set of experiments and corresponding results.

III. FUTURE WORK

This tool represents the introduction of automation of the test case generation activity and respective coverage evaluation within OutSystems applications. Therefore, some limitations are still in place. The future for this tool starts by expanding in terms of the types of nodes it supports for this language, as well as the datatypes it is able to evaluate, as currently the datatypes supported are the basic Integer, Boolean and Strings.

ACKNOWLEDGMENT

The authors would like to thank OutSystems for the support presented throughout the development of this tool.

REFERENCES

- [1] OutSystems <https://www.outsystems.com/>. Last accessed 12 July 2018
- [2] OutSystems: OutSystems - Agile Methodology DataSheet. OutSystems (2010) <https://www.outsystems.com/home/downloadsdetail/25/75/>. Last accessed 11 May 2018
- [3] K. Saravanan and E. Poorna Chandra Prasad: Open Source Software Test Automation Tools: A Competitive Necessity. Scholedge International Journal of Management Development 3(6), 103–110 (2016)
- [4] C. Wenjing and X. Shenghong: A Software Function Testing Method Based on Data Flow Graph. In: 2008 International Symposium on Information Science and Engineering, pp. 28–31. IEEE, Shanghai, China (2008) 10.1109/ISISE.2008.23
- [5] Kshirasagar Naik, Priyadarshi Tripathy: Software testing and quality assurance: theory and practice. 1st edn. Wiley (2008)
- [6] Ammann, Paul and Offutt, Jeff: Introduction to Software Testing, pp.27–51. 1st edn. Cambridge University Press, New York, NY, USA (1999)
- [7] Emher, Mohd and Khan, Farmeena: A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. International Journal of Advanced Computer Science and Applications 3, 12–15 (2012)
- [8] Myers, Glenford J. and Sandler, Corey: The Art of Software Testing. John Wiley & Sons (2004)
- [9] Santos, Ronnie and C. de Magalhaes, Cleiton and Correia-Neto, Jorge and Silva, Fabio and Capretz, Luiz: Would You Like to Motivate Software Testers? Ask Them How. In: Electrical and Computer Engineering Publications, pp. 95–104 (2008) 10.1109/ESEM.2017.16

HTML Document Error Detector and Visualiser for Novice Programmers

Steven Schmoll, Anith Vishwanath, Mohammad Ammar Siddiqui, Boppaiah Koothanda Subbaiah and Caslon Chua

Department of Computer Science and Software Engineering

Swinburne University of Technology

Hawthorn, Victoria, Australia

{100928322, 100049346, 100863533, 101018295}@student.swin.edu.au and cchua@swin.edu.au

Abstract—Learning HTML poses similar challenges as learning conventional programming language by novice programmer. Apart from the HTML validator, there are limited tools to help novice programmer address errors in their HTML code. In this study, we employ visualisation techniques to display the structural and contextual information of the HTML code. We look at condensing and visually representing the important aspects of the HTML code. This is to enable novice programmers gain insights on the HTML code structure and locate any underlying syntax and semantic errors.

Keywords—code visualisation, error detection, computer education

I. INTRODUCTION

There has been limited research to date on the two fundamental languages used in web development, namely HTML and CSS. Like the learning of programming languages by students as novice programmers, HTML and CSS present similar learning challenges such as syntax and runtime errors, including bugs in the form of unintended behaviours [1]. Apart from syntax errors, there are also the semantic errors that novice programmers commit in the HTML code. This is mostly parent-child nesting rules violation, such as element Y is not allowed as a child of element X, missing end tags, or parent element closed with an end tag but has a child element that was not closed [6].

Novice programmers often start their code development by opportunistically modifying sample codes that they found [2]. They may also use online Q&A forums to look for answers to problems that they are trying to solve. Moreover, HTML code fragments found in online Q&A forums can be written in various HTML versions depending on the time the questions were answered. In addition, modern browsers are designed to render HTML code in the best way it possibly can which often ignores errors. With the novice programmer relying on the browser to test their code, they often accept their HTML code as correct based on how it is rendered.

Visualisations can help developers cut through the complexity of code by highlighting patterns, and making the usually invisible software artefacts visible [5]. A review of program visualisation systems for novice programmers found that they are limited to the showing of program animations; and the recent trend is having more user interaction which tend to suggest a positive impact on learning [3]. In this study, we explore the use of visualisation to show the code quality based on the detect errors in the HTML.

II. LEARNING WEB DEVELOPMENT

In an introductory web development unit, students as novice programmers would develop HTML code using a text editor and preview its results with a web browser. This presents an instant gratification, however a closer look at the developed code shows that it often contains a number of logical or semantic errors. Despite emphasising good coding practice and adherence to a specific HTML version, errors are still observed to be prevalent. The current approach is to learn HTML coding through validation, as validation is found to be effective as most of the errors detected were resolved [6]. The study aims to provide novice programmer with an interactive tool that will assist them in analysing HTML code for logical and semantic problems, and highlight these errors through visualisations.

III. DESIGN CONSIDERATION

In our design, we considered visually capturing the structure of the code, and highlighting the location in which the errors were detected. We also looked at the use of colours.

A. Reduced Line Representation

Reduced line representation is a method based on reducing the source code to a point where each keyword is represented using a single pixel. This means code with lines of text can be reduced to rows of pixels, and preserving the indentation, the length, and the colouring. Colour coding the pixel has a good visual effectiveness, as this can be used to represent certain statistical information such as code or error frequency which enable detection of patterns [5].

B. Colour Representation

The stop light colour representation is used in this study to indicate the severity of errors to the novice programmers. These distinct colours which correspond to the severity ratings help the novice programmers understand what type of errors occurred in the HTML code analysed. Various icons are also used to represent items visually. Figure 1 shows that errors are represented in red using a triangular icon, while warnings are represented in amber with a circular icon.



Fig. 1. Stop Light Colour Representation

IV. VISUALISATION APPLICATION

A web-based application was implemented to test the visualisation design on the detected errors. The screen shot shown in Figure 2 summarises the detected error counts and three tab options after the user uploads an HTML file. The three tab options are overview, visualisation and error which allow the user to interactively click on detected errors to show more detailed error information.



Fig. 2. HTML Analyser and Visualiser

A. Overview Presentation

The overview tab allows the novice programmer to interactively browse the code and click on the detected error that is indicated by an error icon at the end of the line as shown in Figure 3.

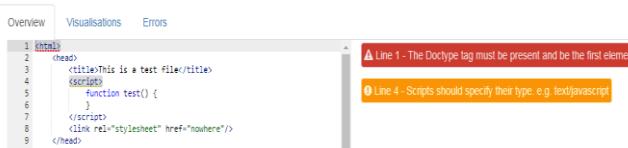


Fig. 3. Overview Tab

B. Visualisation Tab

Under the visualization tab, the code characteristic is presented visually. This enables the novice programmer to quickly assess the quality of the code.

- Indentation Visualisation. Figure 4 shows how the code is structured based on its indentation. While it does not represent the logical structure of the code, it can identify where nesting error may potentially occur. This visualisation is not intended for minified code, but is aimed at novice programmer observing good programming practice.



Fig. 4. Indentation Visualisation

- Error Severity Visualisation. Figure 5 shows the severity of the detected errors presented as a pie chart. This emphasises the quality of the code based on the number of detected errors. A code with no detected errors will not generate a pie chart.

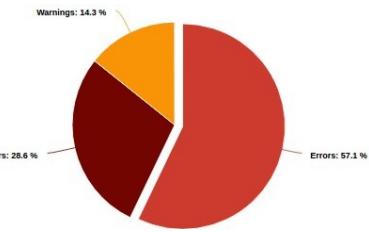


Fig. 5. Error Severity

C. Error Tab

The error tab shown in Figure 6 lists the lines where errors are detected, allowing the novice programmer to interactively look at the code or additional information, such as reasons of the error and how it can be addressed.

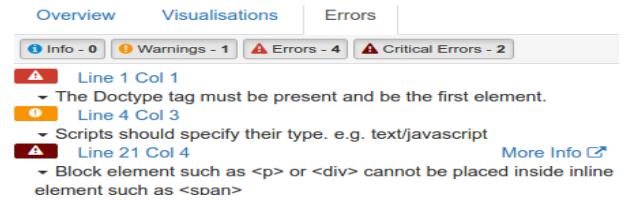


Fig. 6. Error Tab

V. CONCLUSION

In this study, a web-based application to visualise the detected logical errors in HTML code was implemented. A usability test conducted among 10 participants using the System Usability Scale (SUS) yielded a score of 82.5. As next steps, we will incorporate the colour coding scheme into the indentation visualisation and add statistical information to enhance pattern detection. Visualising CSS errors will also be looked at, given that error messages generated by the CSS validator are quite cryptic. Cryptic feedback is known to cause difficulty for novice programmers [4].

Finally, a user study on how visualising detected errors can improve the learning experience of the novice programmers will also be conducted. We aim to have the tool piloted in an introductory web development unit to enable the novice programmer to interactively get feedback on the code that they developed.

REFERENCES

- [1] A. F. Blackwell, "First steps in programming: a rationale for attention investment models," In Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments. pp. 2–10. 2002.
- [2] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Opportunistic programming: writing code to prototype, ideate, and discover," IEEE Software vol.26, no.5, pp. 18-24, 2009.
- [3] J. Sorva, V. Karavirta, and L. Malmi. "A review of generic program visualization systems for introductory programming education," Trans. Comput. Educ. vol. 13, no. 4, Art. 15 (November 2013).
- [4] M. Nienaltowski, M. Pedroni, and B. Meyer, "Compiler error messages: What can help novices?" In Proceedings of the SIGCSE Technical Symposium on Computer Science Education. pp. 168–172, 2007.
- [5] T. Ball and S. Eick, "Software visualization in the large", IEEE Computer, vol. 29, no. 4, pp. 33-43, 1996.
- [6] T. H. Park, B. Dorn, and A. Forte, "An analysis of HTML and CSS syntax errors in a web development course," Trans. Comput. Educ. vol. 15, no. 1, art. 4 (March 2015).

Toward an Efficient User Interface for Block-Based Visual Programming

Yota Inayama Hiroshi Hosobe
Faculty of Computer and Information Sciences
Hosei University
 Tokyo, Japan
 hosobe@acm.org

Abstract—Block-based visual programming (BVP) is becoming popular as a basis of programming education. It allows beginners to visually construct programs without suffering from syntax errors. However, a typical user interface for BVP is inefficient partly because the users need to perform many drag-and-drop operations to put blocks on a program, and also partly because they need to find necessary blocks from many choices. To improve the efficiency of constructing programs in a BVP system, we propose a user interface that introduces three new features: (1) the semiautomatic addition of blocks; (2) the use of a pie menu to change categories of blocks; (3) the focus+context visualization of blocks in a category. We implemented a prototype BVP system with the new user interface.

Index Terms—visual programming, block, user interface

I. INTRODUCTION

We propose a user interface that improves the efficiency of block-based visual programming (BVP). It introduces the following three new features:

- 1) the semiautomatic addition of blocks;
- 2) the use of a pie menu to change categories of blocks;
- 3) the focus+context visualization of blocks in a category.

We implemented a prototype BVP system with the new user interface. Our showpiece is the demonstration of this prototype system.

II. PROBLEMS WITH EXISTING USER INTERFACES

The user interface of Scratch [5] is a representative of those for BVP. Such user interfaces consist mainly of three components, i.e., a set of categories of blocks, a set of blocks in the currently selected category, and a workspace for programming. Users of such interfaces suffer from the following three problems:

- They need to frequently change categories of blocks;
- They need to perform many drag-and-drop operations to construct programs;
- It is often hard for them to find necessary blocks because there are several categories that contain many blocks.

We can explain these problems by using two well-known principles for user interface design. The first principle is Fitts' law [4]. It is able to predict the time length that a user needs to point at a target on a display with a pointing device such

as a mouse. It uses the following formula to predict the time length:

$$T = a + b \log_2 \left(\frac{D}{W} + 1 \right),$$

where D is the distance to the target, W is the size of the target, and a and b are constants that are determined experimentally. Intuitively, this law indicates that, the longer the distance to the target is, or the smaller the size of the target is, the longer time the user needs to point at the target. We can see from this law that users of BVP need time to construct programs with drag-and-drop operations and also to change categories of blocks.

The second principle is Hick's law [8]. It is able to predict the time length that a user needs to select an appropriate item from multiple choices. It uses the following formula to predict the time length:

$$T = a + b \log_2(n + 1),$$

where n is the number of choices, and a and b are constants that are determined experimentally. Intuitively, this law indicates that, the more choices there are, the longer time the user needs to make a decision. We can see from this law that users of BVP need time to select a category of blocks and also to select a necessary block from a category of blocks.

III. OUR USER INTERFACE

To improve the efficiency of constructing programs in a BVP system, we propose a user interface that introduces three new features. The first feature is to enable the user to semiautomatically add a selected block to the visual program. It reduces the time by decreasing the number of the drag-and-drop operations for positioning blocks. In addition, it reduces the mistakes that the user makes to position blocks when the user drops them. In our user interface, the user first selects an existing block on the workspace to indicate that a new block should be added immediately under the selected block. Then the selected block becomes blinking (Figure 1(a)). After this, the user can perform the semiautomatic addition of a new block (Figure 1(b)) just by clicking it on a category of blocks. To enable the successive addition of blocks, such a selected block is automatically updated like a cursor moving in a text editor. The user can also deselect such a block by clicking it.

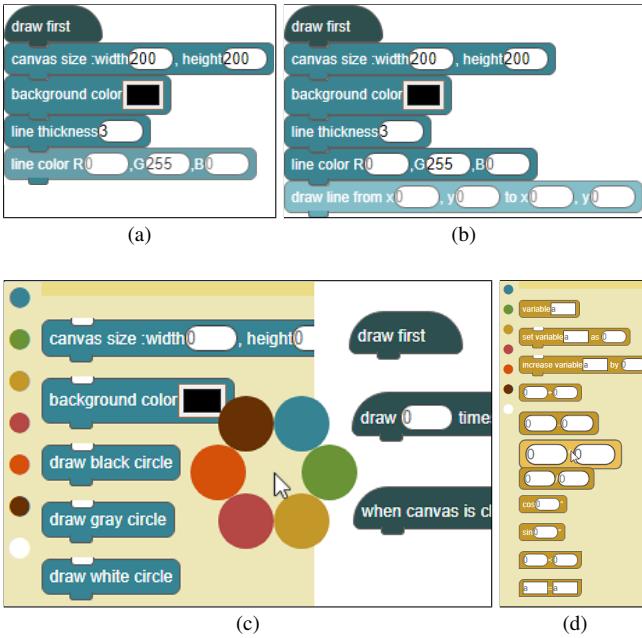


Fig. 1. Our user interface for block-based visual programming.

The second feature is to enable the user to use a pie menu [1] to change categories of blocks. A pie menu is a circular menu where the distance from the center to each menu item is equal and short, which allows the user to more quickly select an item than when using an ordinary linear menu. In addition, it reduces the mistakes that the user makes because the user distinguishes menu items by angles. In our user interface, the user pops up a pie menu around a mouse pointer by pressing the right mouse button (Figure 1(c)). The items in the pie menu correspond to the categories of blocks, and the user can change categories by clicking menu items. In addition, while users of existing interfaces need to click menu items to see the contents of categories, our user interface allows the user to see the content of a different category only by hovering over a menu item, which immediately shows the corresponding category.

The third feature is to enable the user to use the focus+context visualization [3] of blocks. Focus+context visualization simultaneously shows a particular detail and the overview of given information to enable the understanding of the relationship between the important part and the entire structure of the information. In our user interface, the user can change his/her focus by moving the mouse pointer over blocks in a category (Figure 1(d)). This allows the user to more easily recognize blocks around the mouse pointer while viewing the entire category of blocks at the same time. Also, it eases the user to select a block since blocks around the mouse pointer become larger.

IV. IMPLEMENTATION

We implemented a prototype BVP system adopting the user interface that we proposed in the previous section. For this purpose, we extended Kurihara et al.'s BVP system [2], which

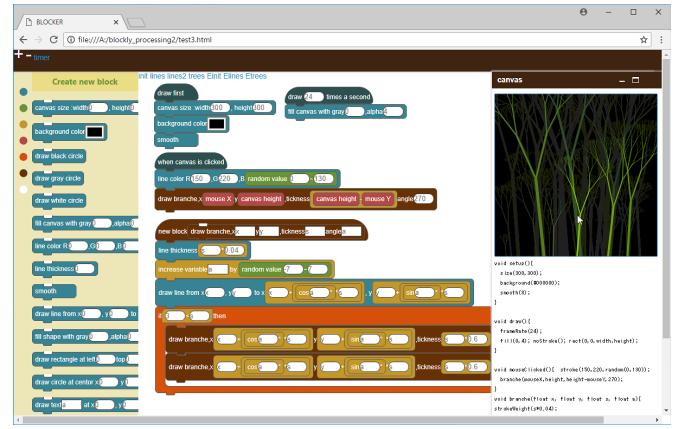


Fig. 2. Our prototype system.

generates programs written in Processing [6]. This system is a Web application written in HTML, JavaScript, and CSS that runs on a Web browser by using the Processing.js [7] library. The user interface of our prototype system consists of three typical components, i.e., a set of categories of blocks, a set of blocks in the currently selected category, and a workspace for programming (Figure 2). There are six categories of blocks that are painted with different colors.

V. CONCLUSIONS AND FUTURE WORK

We proposed a user interface for BVP that introduced three new features. We also implemented a prototype BVP system with the proposed user interface. Our future work includes the experimental evaluation of the performance of the proposed user interface by comparing it with a typical user interface for BVP. Another future direction is to further explore possible features, for example, for enabling users to efficiently entering values in blocks.

ACKNOWLEDGEMENT

This work was partly supported by JSPS KAKENHI Grant Number JP17H01726.

REFERENCES

- [1] D. Hopkins. The design and implementation of pie menus. *Dr. Dobb's J.*, 16(12):16–26, 1991.
- [2] A. Kurihara, A. Sasaki, K. Wakita, and H. Hosobe. A programming environment for visual block-based domain-specific languages. In *Proc. SCSE*, volume 62 of *Procedia CS*, pages 287–296, 2015.
- [3] J. Lamping and R. Rao. The hyperbolic browser: A focus+context technique for visualizing large hierarchies. *J. Visual Lang. Comput.*, 7(1):33–55, 1996.
- [4] I. S. MacKenzie. Fitts' law as a research and design tool in human-computer interaction. *Human-Comput. Interact.*, 7:91–139, 1992.
- [5] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The Scratch programming language and environment. *ACM Trans. Comput. Educ.*, 10(4):16:1–15, 2010.
- [6] C. Reas and B. Fry. Processing: Programming for the media arts. *AI Soc.*, 20(4):526–538, 2006.
- [7] J. Resig. Processing.js, 2008. <https://johnresig.com/blog/processingjs/>
- [8] L. Rosati. How to design interfaces for choice: Hick-Hyman law and classification for information architecture. In *Classification & Visualization: Interfaces to Knowledge*, pages 121–134, 2013.

Human-Centric Programming in the Large - Command Languages to Scalable Cyber Training

Prasun Dewan

*Department of Computer Science
University of North Carolina
Chapel Hill, USA
dewan@cs.unc.edu*

Blake Joyce

*CyVerse
University of Arizona
Tucson, USA
bjoyce3@cyverse.org*

Nirav Merchant

*Data Science Institute
University of Arizona
Tucson, USA
nirav@email.arizona.edu*

Abstract— Programming in the large allows composition of processes executing code written using programming in the small. Traditionally, systems supporting programming in the large have included interpreters of OS command languages, but today, with the emergence of collaborative “big data” science, these systems also include cyberinfrastructures, which allow computations to be carried out on remote machines in the “cloud”. The rationale for these systems, even the traditional command interpreters, is human-centric computing, as they are designed to support quick, interactive development and execution of process workflows. Some cyberinfrastructures extend this human-centricity by also providing manipulation of visualizations of these workflows. To further increase the human-centricity of these systems, we have started a new project on cyber training – instruction in the use of command languages and visual components of cyberinfrastructures. Our objective is to provide scalable remote awareness of trainees’ progress and difficulties, as well as collaborative and automatic resolution of their difficulties. Our current plan is to provide awareness based on a subway workflow metaphor, allow a trainer to collaborate with multiple trainees using a single instance of a command interpreter, and combine research in process and interaction workflows to support automatic help. These research directions can be considered an application of the general principle of integrating programming in the small and large

Keywords—Cyberinfrastructure, workflow, awareness, recommender systems, visual programming

I. INTRODUCTION

By programming in the small, we mean creation of a program whose tasks are executed by a single operating system process, possibly interacting with one or more humans. Programming in the large is creation of a “program” or *process workflow* whose tasks are performed by multiple OS processes, again possibly interacting with one or more humans. Programming in the large, then, relies on programming in the small to create the code executed by the individual processes.

Such programming was first supported by the Unix command interpreter, called the shell. In fact, process composition is perhaps one of the most distinguishing features of Unix, supporting a philosophy in which each application or system program supports one function, and a multi-functional program is created by composing two or more unmodified existing programs. This principle has allowed operating system functionality to be implemented more concisely in Unix than in its predecessor, Multics. For example, a single “grep” program can be composed with an “ls” or “ps” process to search a

directory and process listing, respectively, for a string. Such reuse has also been useful in application programming. For this reason, command languages in successors of Unix have all supported programming in the large.

II. HUMAN-CENTRICITY

Shell-based interactive command interpreters are sufficient but not necessary for programming in the large. It is possible to use, instead, Unix or some other API to programmatically connect processes together using a language (such as C) developed for programming in the small. Arguably, the purpose of command-interpreters is to support programming that is more human-centric – more interactive, collaborative, easier to learn, and/or easier to use for the task at hand. A similar argument can be made, using these characteristics of human-centricity, to argue that traditional command languages are more human-centric than traditional programming languages, whether the latter are used for programming in the small, or for programming in the large given a suitable API.

III. LARGE-SMALL COMMONALITIES

The different degrees of human-centricity in the two programming granularities are both expected and surprising. If the two approaches were equivalent, then there would have been no need to support process composition in command languages. What makes the differences surprising is the argument that traditional programming and command languages are fundamentally the same, with the main difference being that they manipulate ephemeral (in-memory) data (e.g. scalars and arrays) and persistent data (e.g. files and directories), respectively. Heering and Klint [1] have in fact designed a monolingual environment that integrates traditional command, programming, and debugging languages. They have argued that even if such an environment is not practical, an integration exercise can enrich the individual languages. We refer to this principle as the granularity integration principle.

IV. VISUAL PROGRAMMING IN THE LARGE

Both kinds of programming have evolved much since Heering and Klint’s work – especially in increased human-centricity through visual programming. Visual programming in the small has, of course, received much attention in this conference. Figure 1 and 2 illustrate the use of the CyVerse cyberinfrastructure [2], originally called iPlant, to visually manipulate process workflows.

Figure 1 demonstrates visual workflow composition. In Figure 1(a), the user creates a linear process workflow from the programs (FASTX) Trimmer, Clipper, and Quality Filter. In Figure 1(a), the user adds Quality Filter to the pipeline, not by typing its name, but by searching for it based on its name and attributes. Figure 1(b) shows the current programs in the pipeline, which can be edited by adding new programs, or by deleting or reordering existing programs. In Figure 1(c), the user connects the output of a previous program in the pipeline to the input of Quality Filter by choosing the output source from a menu that lists the potential options based on the preceding programs in the pipeline. This form of programming is akin to block-based programming in that in both cases, users can list, select and edit predefined templates.

Figure 2 demonstrates the subway model for visual workflow navigation, which, to the best of our knowledge, does not have a counterpart in programming in the small. In this model, programs in a predefined pipeline are visualized using a subway metaphor. Each predefined pipeline is mapped to a subway line and each program in the pipeline (e.g. Sequence Trimmer) is associated with a subway stop. Segments of the pipeline performing, together, some high-level task (e.g. Assemble Sequence) are put on separate branches. A user clicks on a stop to execute the associated program, and view and manipulate its output, before going to the next stop.

The three forms of programming (in the large) presented here, embody the general principle that a programming system can be made more human-centric, not only through more visualization, but also by making more decisions for the developer, that is, providing more restrictive, and hence easier to learn and use, specification mechanisms. Command languages are more flexible than visual workflow composition, which is, in turn, more flexible than visual workflow navigation. In terms of ease of use and learnability, the reverse order holds among these three programming abstractions.

V. SCALABLE CYBER-LEARNING

Ease of learning, however, is still a major issue in all three forms of programming in the large. A command language is known to be difficult to learn and use. The visual alternatives, on the other hand, are not standard, and ever evolving. Thus, it is important to provide personalized and scalable training for cyberinfrastructure abstractions. These two requirements are apparently conflicting in that there is a limit to the number of trainees a trainer can help. A further complication is that truly scalable training must, unlike the state of the art, be distributed.

We believe the granularity integration principle can be used to significantly improve this situation. Research on programming in the small has developed (a) awareness techniques for monitoring the programming of a relatively large number of novice programmers [3], and (b) automatic recommendation of solutions to novice programmers [4].

We are developing analogs of these techniques for cyberinfrastructures based on the following novel ideas: (1) *Distributed sticky notes*: Support a distributed analog of sticky notes [5] used in face-to-face instruction by trainees to indicate difficulties to trainers. (2) *Subway-based awareness*: When trainees are composing process workflows using command-

languages or visual programming, create, for the trainers, a visualization of the trainee progress using the subway model, having each stop annotated with both summary and detailed information about the progress and difficulties of the trainees. (3) *Shell-based awareness*: Provide a trainer with shell commands to retrieve information about the trainees' progress, which can be more detailed than subway-based awareness, and can include, for instance, a representation of the history of operations executed by the trainees using the shell or its visual alternatives. (4) *Multi-user training shell*: Allow a trainer to collaborate with multiple trainees using a single instance of a command interpreter by injecting trainer commands into the command histories of trainees. (5) *Integration of process and interaction workflow*: Associate each process workflow to be created in a cyber training exercise with an interaction workflow – the kind used to constrain and define the work of employees in a business or government organization – and use this workflow to recommend next steps to those in difficulty.

CyVerse, being a production system, has an active training program, targeted at both domain scientists and students, that extends shell lessons provided by software carpentry [5]. Like software-carpentry, it requires face-to-face interaction with trainees. We propose to use our technical innovations to make this personalized training program distributed and more scalable, which will yield field data regarding their use. In addition, our planned evaluation includes controlled comparative lab studies

How these ideas may be fleshed out is a matter of research and is likely to benefit from conversations with conference attendees, who, in turn, would learn about the state of the art in visual programming in the large, its relationship to visual programming in the small, granularity integration, and our thoughts on using this principle to advance cyber training.

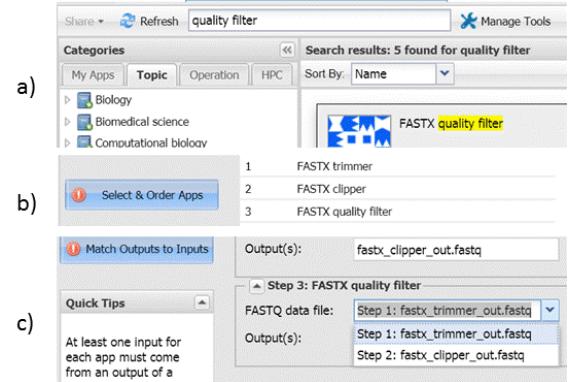


Fig. 1. Visually Creating a Workflow in Cyverse Discovery

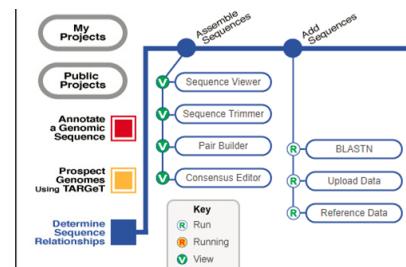


Fig. 2. Manipulating a Predefined Workflow in CyVerse DNA Subway

REFERENCES

- [1] Heering, J. and P. Klint, Towards Monolingual Programming Environments. ACM TOPLAS, 1985. 7(2).
- [2] Merchant, N., E. Lyons, S. Goff, M. Vaughn, D. Ware, D. Micklos, and P. Antin, The iPlant collaborative: cyberinfrastructure for enabling data to discovery for the life sciences. PLoS biology ce, 2011. 14(1).
- [3] Guo, P.J. Codeopticon: Real-Time, One-To-Many Human Tutoring for Computer Programming. in ACM Symposium on User Interface Software and Technology (UIST). 2015.
- [4] Thomas W. Price, Y.D., Tiffany Barnes. Generating Data-driven Hints for Open-ended Programming. in EDM. 2016.
- [5] Carpentry, S. Instructor Training. 2016; Available from: <http://swcarpentry.github.io/instructor-training>.

Visual Knowledge Negotiation

Alan Blackwell

*Computer Laboratory
University of Cambridge
Cambridge, UK
Alan.Blackwell@cl.cam.ac.uk*

Luke Church

*Computer Laboratory
University of Cambridge
Cambridge, UK
luke@church.name*

Matthew Mahmoudi

*Department of Sociology
University of Cambridge
Cambridge, UK
mm2134@cam.ac.uk*

Mariana Mărășoiu

*Computer Laboratory
University of Cambridge
Cambridge, UK
mariana.marasoiu@cl.cam.ac.uk*

Abstract—We ask how users interact with ‘knowledge’ in the context of artificial intelligence systems. Four examples of visual interfaces demonstrate the need for such systems to allow room for negotiation between domain experts, automated statistical models, and the people who are involved in collecting and providing data.

Index Terms—intelligent interfaces, visualisation, knowledge negotiation

I. WHY WE NEED KNOWLEDGE NEGOTIATION

Philip Agre’s classic critique of Artificial Intelligence research articulates a key problem in the mechanisation of knowledge, which he formulates as the “discursive practice” of AI research [1]. This is best summarised in his own words:

AI is a discursive practice. A word such as planning, having been made into a technical term of art, has two very different faces. When a running computer program is described as planning to go shopping, for example, the practitioner’s sense of technical accomplishment depends in part upon the vernacular meaning of the word [...] On the other hand, it is only possible to describe a program as “planning” when “planning” is given a formal definition in terms of mathematical entities or computational structures and processes. [...] This dual character of AI terminology — the vernacular and formal faces that each technical term presents — has enormous consequences for the borderlands between AI and its application domains.

Recent critique of machine learning methods, in Cheney-Lippold’s “We Are Data” [2], identifies a related issue in machine learning. He proposes that the named categories and labels fundamental to supervised machine learning should always be placed in quotation marks, in order to avoid the implication that these names correspond to the “vernacular face” (in Agre’s terms) of concept names outside of the statistical model. For example, Cheney-Lippold notes that his own Google profile identifies him as being “female” (through statistical analysis of his online behaviour) when this is not true. Nevertheless, the statistical observations of Cheney-Lippold as a “female” customer within Google’s models may be useful data for their advertisers, and may be a good prediction of Cheney-Lippold’s future purchases. But when

Case studies funded by Africa’s Voices Foundation, Boeing, BT, EPSRC and the Health Foundation

making use of this fact it is important to remember that this model-label, although potentially useful, is not true.

Building intelligent systems to be useful in some application domain requires constant attention to the necessary dual character of the “knowledge” encoded in the system, and the vernacular language of the user. Where statistical models result in interactive visual languages, we have a critical design problem. Should the visual language correspond to one type of knowledge (which?), or to both?

We claim that visual interaction with intelligent algorithms must be designed in order to allow *negotiation* between the user and the inferred statistical “knowledge”. In summary, visual languages support negotiation of knowledge, *because they are not linguistically over-determined*.

II. VISUAL DESIGN FOR KNOWLEDGE NEGOTIATION

We illustrate this theoretical concern with four practical case studies, supported by visual interfaces as seen in Figures 1, 2, 3 and 4. (Longer descriptions of these case studies are being presented at a satellite workshop of this conference, on Designing Technologies to Support Human Problem Solving [3]).

Each of these four systems is designed for use by a specific class of domain expert — police analysts (Fig 1), business analysts (Fig 2), research translators (Fig 3) and hospital

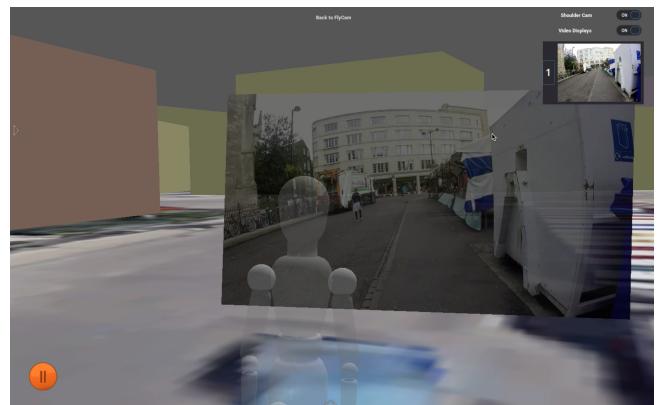


Fig. 1. In ForensicMesh, computer vision algorithms locate video from a body worn camera in a city location, but emphasising the subjective viewpoint of the person wearing it by rendering that person’s body in the foreground, so that the user can interpret this “objective” digital evidence within a subject context.

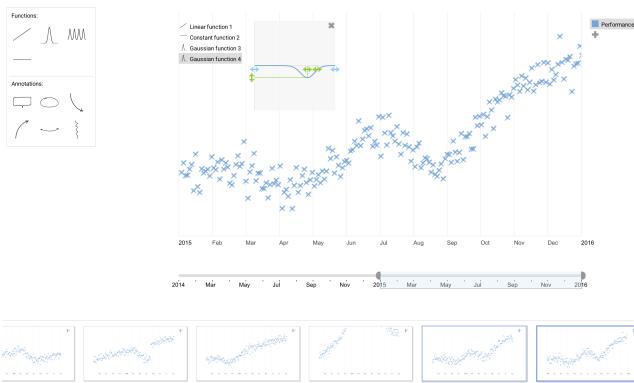


Fig. 2. In SelfRaisingData, a statistical model of unseen data is synthesised by a business analyst as a way of formulating research questions from a user perspective.

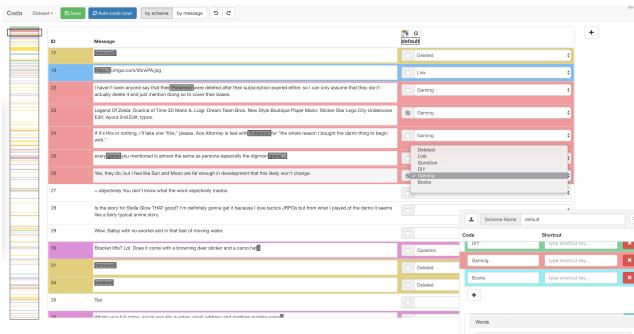


Fig. 3. In Coda, Somali translators classify SMS messages relating to public health, with semi-automated labels negotiated through varying shades of the category colours.¹

clinicians (Fig 4). In each case, a model has been constructed on the basis of data originally acquired from human sources. A statistical model, more or less complete and more or less accurate, has been created on the basis of that data. And in each case, the domain expert who interacts with the system has a richer, more sophisticated and more complete understanding of the context than has been embedded in the model.

That expert understanding extends beyond critical evaluation of the predictive power of the statistical models — it also extends to critical understanding of the data from which the model has been created, and of the human agency through which the data was captured. We therefore try to avoid system designs in which models are trained with a pre-defined set of labels that might be liable to simple acceptance as the full and complete truth — so in Coda (Fig 3), the set of labels can always be expanded, redefined, or replaced with other sets.

We also try to highlight the human origins of apparently mechanical data acquisition, for example in ForensicMesh (Fig 1) we render a human figure into the scene, representing

¹Since the data Coda is used with is usually sensitive, the data in this screenshot is a sample from the Reddit comment data available on Google BigQuery (https://bigquery.cloud.google.com/table/fh-bigquery:reddit_comments)

the police officer who was wearing a body-worn camera from which video was collected.

In the extreme case of SelfRaisingData (Fig 2), we proceed with no data at all, giving expert analysts the opportunity to negotiate far further down the ‘supply chain’ of statistical inference by *creating* a data set. This has no objective status at all, in that no data exists, but provides a basis for negotiating the model that might be created.

ICUMAP (Fig 4) also subverts the conventional visual language of statistics by creating a clustering algorithm that is not a simple dimension reduction of a multivariate space, but modifies the t-SNE distance metric to allow the narrative of a journey (through placing successive time-point samples nearby), and explicitly reflecting the clinicians’ prior expectation (by weighting clusters to represent the most salient clinical category of surgical procedure). These allow clinicians to reason ‘outward’ from their own knowledge to explore statistical similarities beyond the ‘obvious’ (to clinicians) prior expectations.

To conclude, these design case studies demonstrate how visual languages can support negotiation of knowledge, where statistical terminology fails to distinguish between model and application.

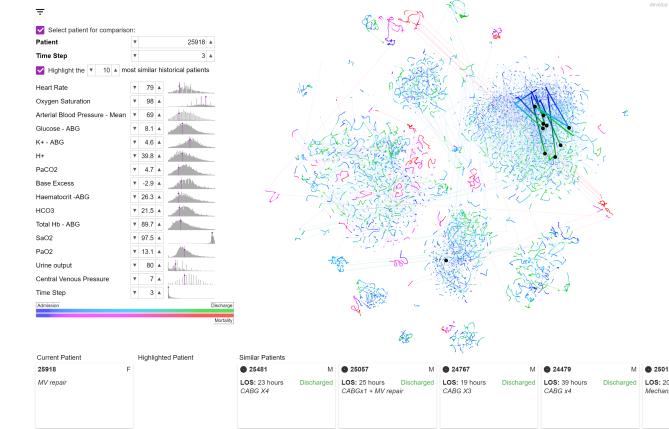


Fig. 4. In ICUMAP, the outcomes of post-surgery intensive care are visualised as trajectories toward discharge (green) or mortality (red), so that clinicians can assess typicality or risk of new cases in relation to precedent, but without relinquishing judgment.

REFERENCES

- [1] P. E. Agre, “Toward a Critical Technical Practice: Lessons Learned in Trying to Reform AI,” in *Bridging the Great Divide: Social Science, Technical Systems, and Cooperative Work*, L. S. Les Gasser and G. B. Bill Turner, Eds. Erlbaum, 1997.
- [2] J. Cheney-Lippold, *We are data: Algorithms and the making of our digital selves*. NYU Press, 2017.
- [3] A. Blackwell, L. Church, M. Jones, R. Jones, M. Mahmoudi, M. Marasou, S. Makins, D. Nauck, K. Prince, A. Semrov, A. Simpson, M. Spott, A. Vuylsteke, and X. Wang, “Computer says ‘don’t know’ - interacting visually with incomplete AI models,” in *Designing Technologies to Support Human Problem Solving Workshop - A Workshop in Conjunction with VL/HCC 2018*, 2018.

A Modelling Language for Defining Cloud Simulation Scenarios in RECAP Project Context

Cleber Matos de Moraes

Universidade Federal da Paraiba

Joao Pessoa, Paraiba

cmorais@cchla.ufpb.br

Patricia Endo

*Irish Centre for Cloud Computing and Commerce (IC4)**Dublin City University (DCU)*

Dublin, Ireland

patricia.endo@dcu.ie

Sergej Svorobej

*Irish Centre for Cloud Computing and Commerce (IC4)**Dublin City University (DCU)*

Dublin, Ireland

sergej.svorobej@dcu.ie

Theo Lynn

*Irish Centre for Cloud Computing and Commerce (IC4)**Dublin City University (DCU)*

Dublin, Ireland

theo.lynn@dcu.ie

Abstract—The RECAP is a European Union funded project that seeks to develop a next-generation resource management solution, from both technical and business perspectives, when adopting technological solutions spanning across cloud, fog, and edge layers. The RECAP project is composed of a set of use cases that present highly complex and scenario-specific requirements that should be modelled and simulated in order to find optimal solutions for resource management. Due use cases characteristics, configuring simulation scenarios is a high time consuming task and requires staff with specialist expertise.

This work proposes the Simulation Modelling Language (SML), a domain-specific modelling language based on the Model-Driven Development (MDD) paradigm, that assists a cloud infrastructure manager to plan and generate simulations faster and using a friendly graphical interface.

I. INTRODUCTION

Internet of Things (IoT) are transforming how society operates and interacts with each other. However, the relatively small size and heterogeneity of connected edge devices typically results in limited storage and processing capacity, and consequential issues regarding reliability, performance, and security. Some of these IoT issues can be mitigated by integrating fog and cloud computing.

In order to mitigate current large-scale cloud/fog/edge system issues (such as heterogeneity, cost, energy efficiency, and high availability), the RECAP (Reliable Capacity Provisioning and Enhanced Remediation for Distributed Cloud Applications) project¹, a European Union funded project, seeks to develop a next generation cloud/fog/edge resource provisioning and remediation solution via targeted research advances in cloud infrastructure optimization, simulation and

This work is partly funded by the Irish Centre for Cloud Computing and Commerce, a Enterprise Ireland/IDA Technology Centre, and by the European Unions Horizon 2020 Research and Innovation Programme through RECAP (<http://www.recap-project.eu>) under Grant Agreement Number 732667.

¹<https://recap-project.eu/>

automation [1]. RECAP is producing a number of distinct tools designed to operate together, including the RECAP Simulator Framework.

II. RECAP SIMULATION FRAMEWORK

The RECAP Simulation Framework enables reproducible and controllable experiments, aiding in identifying targets for the deployment of software components and in optimizing deployment choices prior to the actual deployment in a real cloud environment (see Figure 1).

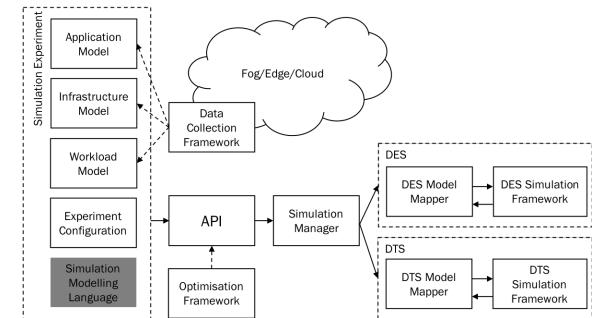


Fig. 1. RECAP Simulation Framework high level design with the proposed Simulation Modelling Language (SML)

The RECAP Simulation Framework requirements are based on the description of the use cases compiled by the RECAP project partners. These use cases describe the challenges the industry faces today from both technical and business perspectives when adopting technological solutions spanning across cloud, fog, and edge layers. Use cases include cloud infrastructure and network management, big data analytics, IoT in smart cities, virtual content distribution networks (vCDNs) and network function virtualisation (NFV), as detailed in [2].

In order to set-up a simulation, it is necessary *(a)* to have a scenario configuration file (that describes, for instance, virtual and physical machines, and network elements) and *(b)* to

define observable metrics (such as memory consumption on a physical machine).

The RECAP Simulation Framework user, who wishes to evaluate a given scenario, must map the real configuration of the use case with a semantic configuration. Unfortunately, each use case can be characterized as both highly complex and scenario-specific requiring unique simulation configurations and measurements. As such, these configurations are time consuming and require experienced staff with specialist expertise.

To reduce simulation configuration time, associate effort and the need for specialist personnel, this paper proposes the Simulation Modelling Language (SML) based on the Model-Driven Development (MDD) paradigm.

III. SIMULATION MODELLING LANGUAGE (SML)

The Simulation Modelling Language (SML) is a domain-specific language focused on the configuration of a wide range of use cases simulations for the RECAP project. The main objective is to assist a cloud infrastructure manager to generate simulations faster and easier using a graphical user interface.

Figure 2 presents the entities that can be used to configure a simulation experiment. The user equipment is the entity that makes a request to a service; depending on the user type, the service can be classified as user or control plane. A service can be any virtual application deployed in a physical machine that has a location and is related to a network tier in the hierarchical topology of the system.

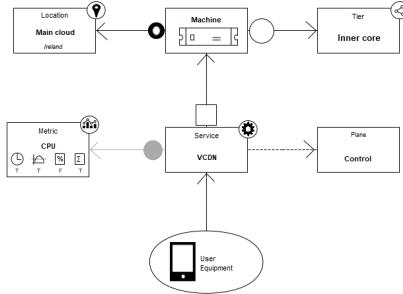


Fig. 2. SML entities

Beyond these entities, the language also offers an entity that represents the metrics one can use to set the simulation experiments. For metrics, one can choose computational resource consumption (CPU, memory, and storage), network resource (e.g. bandwidth), and application specific metrics (e.g. cache hit, cache miss). The metrics can be shown as sum, average, percentage or time series. Figure 3 describes how entities can be connected in the SML.

For illustration purposes, consider the case of vCDNs. Figure 4 depicts a possible simulation configuration. In this example, the vCDN cache hit and cache miss are the monitored metric, and the computational metrics are measured only in the machines located at MSAN and Inner Core tiers; and the network utilisation is monitored in all network tiers. If needed, other configurations are also allowed by the SML: for instance,

Connector	From	Action	To
	User Machine Service Switch	Sends/forwards request	Service Switch
	Service	Is placed in	Plane
	Machine Service Tier	Is measured by	Metric
	Service	Is mapped in a	Machine
	Machine Switch	Is connected to a	Tier
	Machine Switch	Is physically located at	Location

Fig. 3. SML connectors

one can set vCDN applications only at Inner Core tier, and all user requests will be forwarded by the switches through the network.

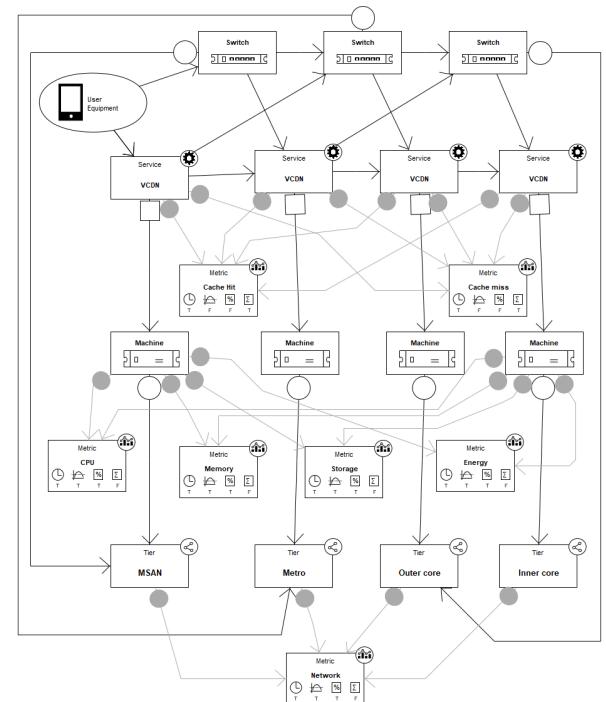


Fig. 4. RECAP Simulation Framework high level design

As future works, we plan to validate the SML with all RECAP use cases, and also generate the simulation scripts that can be automatically used by the RECAP Simulation Framework to evaluate the use cases.

REFERENCES

- [1] P.-O. Östberg, J. Byrne, P. Casari, P. Eardley, A. Fernandez Anta, J. Forsman, J. Kennedy, P. Le Duc, M. Noya Mariño *et al.*, “Reliable capacity provisioning for distributed cloud/edge/fog computing applications,” in *Networks and Communications (EuCNC), 2017 European Conference on*. Universität Ulm, 2017.
- [2] J. Domaschka, “Deliverable 3.1. initial requirements,” RECAP Project, Tech. Rep.

A Vision for Interactive Suggested Examples for Novice Programmers

Michelle Ichinco

*Department of Computer Science
University of Massachusetts Lowell
Lowell, MA, USA
michelle_ichinco@uml.edu*

Abstract—Many systems aim to support programmers within a programming context, whether they recommend API methods, example code, or hints to help novices solve a task. The recommendations may change based on the user’s code context, history, or the source of the recommendation content. They are designed to primarily support users in improving their code or working toward a task solution. The recommendations themselves rarely provide support for a user to interact with them directly, especially in ways that benefit the knowledge or understanding of the user. This poster presents a vision and preliminary designs for three ways a user might learn from interactions with suggested examples: describing examples, providing detailed relevance feedback, and selective visualization and tinkering.

Index Terms—Novice programmers, interactive suggestions, example code

I. INTRODUCTION

Many systems suggest example code or support novices in finding example code relevant to their programs, both in task-based contexts as well as when programmers design their own projects [1]–[3]. Beyond a description or highlighting of the relevant elements, the examples typically provide little support for learning [4], [5]. Many novice programmers begin to learn independently, in non-task contexts, like by creating their own app, website, or game. Novices in these contexts thus need more support from suggestion-based systems to actually learn from examples. Research in cognitive load theory provides approaches for increasing learning gains from educational material. This poster presents designs for three interaction techniques for suggested examples based on cognitive load theory.

II. COGNITIVE LOAD THEORY

Cognitive load theory is a theory often used in the design of instructional material in order to support learning [6]. It is often associated with ‘worked examples’, which are examples with worked out steps, usually for topics like mathematics or physics. Humans have limited cognitive load to spend at any point in time. Cognitive load theory provides methods of reducing extraneous cognitive load, which interferes with learning, and increasing germane cognitive load, which supports deep learning. This vision incorporates three elements of cognitive load theory into the design of interaction methods: self-explanation, multiple examples, and fading.

Self-explanation causes learners to produce explanations of new material and relate those explanations to the general principles being taught [7]. This process deepens learners’ understanding of the new material. Recent work has shown that self-explanation can encourage learners to create labels for programming examples [8], [9]. Ideally, encouraging novices to author descriptions of code examples will result in the benefits of effective self-explanation.

Self-explanation can be especially helpful for the comparison of multiple examples. Research has found that providing multiple worked examples can help learners [10]. However, Catrambone and Holyoak found that multiple examples only support learners in problem solving when the learners explore the similarities between the examples [11]. Having learners explain the relevance or lack of relevance between their code and examples, will likely have similar effects to the presentation of multiple worked examples combined with self-explanation.

While self-explanation increases germane cognitive load, faded worked examples reduce the amount of new information a learner needs to deal with at one time. Fading involves a sequence of worked examples. In each subsequent worked example, support is removed [12]. Thus, fading supports learners by reducing the extraneous cognitive load. Researchers have shown that faded worked examples can be effective for programming [13]. Our third interaction method, selective visualization and tinkering, aims to approximate fading by enabling novices to focus on smaller pieces of an example, rather than attempting to understand how the entire example works at one time.

III. SUGGESTION INTERACTION TYPES

This poster presents three potential interaction methods: 1) describing code examples, 2) providing detailed relevance feedback, and 3) selective visualization and tinkering. Each section describes why this interaction method should support learning and why users will likely be motivated to participate in these interactive activities.

IV. DESCRIBING CODE EXAMPLES AND SELECTING APPROPRIATE DESCRIPTIONS

Having learners describe code examples and select appropriate descriptions should elicit self-explanation of the provided examples. In order to describe an example or select a correct description, the learner will have to figure out how it works.

Other tools for learning have had learners successfully label videos [14], math problems [15], and programming worked examples [8]. These studies support the value of this method of eliciting self-explanation, but evaluate the method in controlled studies where that is a primary task. I hypothesize that this type of approach would also work in the midst of a programming task where a user can choose whether or not to participate in the example labeling. One way to motivate users to author these descriptions may be by telling them that their description will help other users. Many programmers choose to help each other, such as in the Stack Overflow forum [16]. If users are motivated by helping each other, they will likely try to author or select the best description they can.

This process of describing code examples will likely encourage learners to think more deeply about examples. We also want to encourage learners to think deeply about why they received feedback and how it relates to their own code by encouraging them to provide detailed relevance feedback.

V. PROVIDING DETAILED RELEVANCE FEEDBACK

Thinking critically about the relevance of a suggested example to a user's code will likely deepen their understanding of the example and their code. Compared to a quick up or down vote like in many existing systems, providing detailed feedback would hopefully encourage a learner to spend time thinking about how their code is related or not related to the suggested example. As a side benefit, these descriptions can also help the system designers to improve the relevance of suggestions or evaluate a learner's understanding. Leveraging this fact may encourage learners to provide detailed descriptions, as the better their descriptions are, the better the support system can help them.

VI. SELECTIVE VISUALIZATION AND TINKERING

Current example code suggestions do not typically enable users to tinker with or explore examples without inserting them into the user's code. Some allow execution of the entire code snippet, along with a visualization of the entire code snippet output, but do not allow changes or selecting a specific part of the code to execute [4]. When watching the execution of a whole code snippet, it might be hard, especially for a novice, to determine which elements of code have which effect. Enabling a user to tinker with an example without inputting it into their code might enable them to better understand each element of a code example before they try to implement it themselves. This could prevent novices from creating new errors when using new code.

VII. FUTURE WORK

This poster presents a vision and preliminary designs for interactive suggested examples. The ideas presented require iteration and evaluation with users. User studies will reveal whether users' motivations match the provided interaction opportunities and whether these interaction methods improve learning.

REFERENCES

- [1] J. Cao, Y. Riche, S. Wiedenbeck, M. Burnett, and V. Grigoreanu, "End-user mashup programming: through the design lens," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2010, pp. 1009–1018. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1753477>
- [2] T. W. Price, Y. Dong, and D. Lipovac, "iSnap: towards intelligent tutoring in novice programming environments," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, 2017, pp. 483–488.
- [3] M. Ichinco and C. Kelleher, "Towards block code examples that help young novices notice critical elements," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct. 2017, pp. 335–336.
- [4] M. Ichinco, W. Y. Hnin, and C. L. Kelleher, "Suggesting API Usage to Novice Programmers with the Example Guru," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, ser. CHI '17. New York, NY, USA: ACM, 2017, pp. 1105–1117. [Online]. Available: <http://doi.acm.org/10.1145/3025453.3025827>
- [5] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do: suggesting solutions to error messages," in *Proc. 28th int. conf. on Human factors in computing systems*, 2010, pp. 1019–1028.
- [6] J. Sweller, "Cognitive load theory, learning difficulty, and instructional design," *Learning and instruction*, vol. 4, no. 4, pp. 295–312, 1994.
- [7] M. T. Chi, M. Bassok, M. W. Lewis, P. Reimann, and R. Glaser, "Self-explanations: How students study and use examples in learning to solve problems," *Cognitive science*, vol. 13, no. 2, pp. 145–182, 1989.
- [8] B. B. Morrison, L. E. Margulieux, and M. Guzdial, "Subgoals, context, and worked examples in learning computing problem solving," in *Proceedings of the eleventh annual international conference on international computing education research*. ACM, 2015, pp. 21–29.
- [9] B. B. Morrison, L. E. Margulieux, B. Ericson, and M. Guzdial, "Subgoals help students solve Parsons problems," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, 2016, pp. 42–47.
- [10] R. K. Atkinson, S. J. Derry, A. Renkl, and D. Wortham, "Learning from examples: Instructional principles from the worked examples research," *Review of educational research*, vol. 70, no. 2, pp. 181–214, 2000. [Online]. Available: <http://rer.sagepub.com/content/70/2/181.short>
- [11] R. Catrambone and K. J. Holyoak, "Overcoming contextual limitations on problem-solving transfer," *Journal of Experimental Psychology: Learning, Memory, and Cognition*, vol. 15, no. 6, p. 1147, 1989.
- [12] A. Renkl, R. K. Atkinson, and C. S. Gro\`s se, "How fading worked solution steps worksa cognitive load perspective," *Instructional Science*, vol. 32, no. 1-2, pp. 59–82, 2004.
- [13] S. Gray, C. St Clair, R. James, and J. Mead, "Suggestions for graduated exposure to programming concepts using fading worked examples," in *Proceedings of the third international workshop on Computing education research*. ACM, 2007, pp. 99–110.
- [14] S. Weir, J. Kim, K. Z. Gajos, and R. C. Miller, "Learnersourcing subgoal labels for how-to videos," in *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*. ACM, 2015, pp. 405–416.
- [15] J. J. Williams, J. Kim, A. Rafferty, S. Maldonado, K. Z. Gajos, W. S. Lasecki, and N. Heffernan, "Axis: Generating explanations at scale with learnersourcing and machine learning," in *Proceedings of the Third (2016) ACM Conference on Learning@ Scale*. ACM, 2016, pp. 379–388.
- [16] "Stack overflow." [Online]. Available: <http://stackoverflow.com>

An Exploratory Study of Web Foraging to Understand and Support Programming Decisions

Jane Hsieh¹, Michael Xieyang Liu², Brad A. Myers², and Aniket Kittur²

¹ Department of Computer Science
Oberlin College
Oberlin, OH, USA
jhsieh@oberlin.edu

² Human Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, PA, USA
{xieyangl, bam, nkittur}@cs.cmu.edu

Abstract— Programmers consistently engage in cognitively demanding tasks such as sensemaking and decision-making. During the information-foraging process, programmers are growing more reliant on resources available online since they contain masses of crowdsourced information and are easier to navigate. Content available in questions and answers on Stack Overflow presents a unique platform for studying the types of problems encountered in programming and possible solutions. In addition to classifying these questions, we introduce possible visual representations for organizing the gathered information and propose that such models may help reduce the cost of navigating, understanding and choosing solution alternatives.

Keywords— *information-foraging, exploratory programming, decision-making*

I. INTRODUCTION

Programming is not solely comprised of coding. Developers spend a significant amount of time foraging for and making sense of the information they need before making changes to a software system [1]. Previous empirical studies have revealed that as much as 35-50% of programming time is spent exploring and seeking information [1-2]. During this time, developers must engage in a variety of cognitive activities such as understanding unfamiliar pieces of code and deciding how to modify existing pieces of software, as well as higher-level decisions such as choosing which APIs to use.

These exploratory activities are *foraging tasks* [4] where developers seek to collect information about the different options or ways of implementing desired programs. Often, the programmer attempts to achieve more than just gathering such content, they also engage in *sensemaking* [4] so that the newly acquired knowledge can be utilized to make decisions about how to implement or amend their own code. In this study, we use available content from Stack Overflow posts to gain insight about the *categories of problems* that programmers experience and the *types of information* that guides their sensemaking and decision-making processes.

We began by manually analyzing a preliminary sample of 92 questions and classifying them into four broad categories of inquiries: methodological (31% of the questions), debugging (29%), conceptual (20%), and concept-specific (20%). These categories closely resemble the types previously identified by a clustering method from machine learning [5]. Next we present the *comparison table* for representing questions involving decision-making tasks in some of these

categories. When analyzing the first sample, we noticed that many of the questions contained alternative solutions for solving similar (if not completely equivalent) problems. Furthermore, these additional answers (supplementary to the accepted answer) receive upvotes from the community for the different criteria that they each fulfill. Building off of this observation, we identified that a significant portion of Stack Overflow questions relate to decision making tasks and can therefore be represented in the form of a comparison table.

To verify this hypothesis, we sampled a larger set of questions and attempted to represent the question and answer posts with a table view. The sample query was fine-tuned to capture not only the individually popular questions, but also the “long tail” questions that collectively make up a significant portion of the search traffic [6].

Our results showed that the comparison table is a suitable way of representing information from about half of the Stack Overflow question posts. The usefulness of the comparison table encourages the development and construction of assistive tools utilizing these theoretical models.

II. SAMPLING METHOD AND RESULTS

A. Preliminary Sampling

In order to find an appropriate sample of questions with diverse topics, we used a variety of search queries. Readily available are the built-in Stack Overflow filters: *relevance*, *newest*, *votes*, and *active*. However, to obtain any results, these filters must be accompanied by a nonempty search term. There also exists filters that do not require specific search terms: *interesting*, *featured*, *hot*, *week* and *month*. Table I shows the preliminary search queries and the number of questions sampled from each query result.

TABLE I. PRELIMINARY SAMPLING QUERIES

Queries	Questions
“how to answers:10” ^a with <i>active</i> filter	21
“which should views:500000” ^b	20
<i>Hot</i> filter (“hottest” questions today)	20
<i>Month</i> filter	19
“how to” with <i>votes</i> filter	13

^a The tag “answers:10” results in only questions with 10 or more answers

^b Similarly, “views:500000” filters out questions with less than 500000 views

Since these questions were manually categorized, the classification of the sample questions may be subject to bias. However, the question categories were created by the researcher before encountering the categories found in the clustering method utilized by Allamanis and Sutton [5], and to our surprise there was a correspondence between four of the broad categories extracted from their method and ours:

Methodological: questions where the programmer forages for methods or code snippets to achieve a set of specifications.

Debugging: questions with specific context such as error messages.

Conceptual inquiries: abstract questions about concepts not explained comprehensively in the API documentation.

Concept-specific: questions where the forager seeks to understand how to use particular methods or commands.

When forming these categories and classifying questions into their respective types, we noticed that most of the *methodological* and *concept-specific* questions (51% of all posts) contain answers with multiple options. Each solution is valuable to the community due to a unique set of criteria that they may fulfill. Frequent criteria include factors such as performance/speed, compatibility (with libraries, browser and language versions, etc.), and readability.

Such questions and their multitude of crowdsourced answers suggests that half or more of the questions posted can be visually represented with a *comparison table* - where rows consist of options and columns display the various criteria. For each criteria that an option fills, their intersecting cell can be marked to symbolize relevance. This visualization may help users to not only understand the different options, but also guide them in choosing the one that is most appropriate to their specific situation. To evaluate the practicality of this representation, we needed a larger sample of questions to test the proportion of questions posted that can be represented with the comparison table.

B. Test of Model using Refined Sample Queries

We utilized two new queries to test the usefulness of the comparison table. This stage takes advantage of the advanced search filters of Stack Overflow and how to use them without a search term. Hence, the first 50 questions were collected using the query “*is:question views:2360000*”, which asks for all questions with 2.36 million or more views (there were exactly 50 as of 7/12/18).

However, choosing questions with the most views can be considered cherry-picking since the most popular questions may only be representative of a narrow set of topics, and indeed we do observe a high correlation between popular questions and their compatibility with the comparison table. It is important we consider not only this specific set of popular topics, since previous research has indicated that only a small portion of the search interests from individual information seekers lie within the most popular questions. The remaining interests of the population makes up the majority of topics in

the “long tail” – topics which are less frequently viewed in total, but collectively they cause a significant portion of the total search traffic [5][6].

To sample questions that belong more to the “long tail”, we composed a decidedly restricted query: “*is:question created:2018-06-15 answers:3*” - to find questions with three or more answers that were asked on a particular day. A total of 90 questions were assessed with this query, and we found that 88% of the most viewed questions naturally fit well with a comparison table. And in the final sample, we discover that approximately half (49%) of the questions were representable with the proposed table. This result is consistent with the hypothesis that questions involving decisions (51% of both samples) can be depicted in a tabulated format.

III. RELEVANCE AND IMPLICATIONS

This study is intended to motivate the design of mental models such as the comparison table to reduce the cost of collecting and organizing information for foragers. Many tools can be built based upon proposed designs, and our research group is in the process of developing a web-browsing tool that utilizes the comparison table. Future work is needed to test that such tools are useful to developers as they forage for information in real-life programming contexts. It would also be interesting to study to what extent decision questions like these are common in other domains besides programming, and if our proposed tools could help in those situations as well.

ACKNOWLEDGMENT

This research is supported in part by the CMU REUSE program, funded by NSF grant CCF-1560137 and in part by NSF grant CCF-1814826. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the NSF.

REFERENCES

- [1] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. 2006. “An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks,” IEEE Trans. Softw. Eng. 32, 12 (December 2006), 971-987.
- [2] D. J. Piorkowski, S. D. Fleming, I. Kwan, M. M. Burnett, C. Scaffidi, R. K.E. Bellamy, and J. Jordahl. 2013. “The whats and hows of programmers’ foraging diets,” Proceedings CHI ’13. 3063-3072.
- [3] M. Allamanis and C. Sutton. 2013. “Why, when, and what: analyzing stack overflow questions by topic, type, and code,” Proceedings of the IEEE Conf. on Mining Software Repositories (MSR ’13). 53-56.
- [4] K. Fisher, S. Counts, and A. Kittur. 2012. “Distributed sensemaking: improving sensemaking by leveraging the efforts of previous users,” In Proceedings CHI ’12. 247-256.
- [5] M. S. Bernstein, J. Teevan, S. Dumais, D. Liebling, and E. Horvitz. 2012. “Direct answers for search queries in the long tail.” In Proceedings CHI ’12. 237-246.
- [6] B. Evans and S. Card. 2008. “Augmented information assimilation: social and algorithmic web aids for the information long tail,” In Proceedings CHI ’08. 989-998.

Graphical Visualization of Difficulties Predicted from Interaction Logs

Duri Long
 Georgia Institute of Technology
 Atlanta, GA, USA
 duri@gatech.edu

Kun Wang
 UNC-Chapel Hill
 Chapel Hill, NC, USA
 wangkl@cs.unc.edu

Jason Carter
 Cisco Systems-RTP
 Raleigh, NC, USA
 jasoncartercs@gmail.com

Prasun Dewan
 UNC-Chapel Hill
 Chapel Hill, NC, USA
 dewan@cs.unc.edu

Abstract— Automatic detection of programmer difficulty can help programmers receive timely assistance. Aggregate statistics are often used to evaluate difficulty detection algorithms, but this paper demonstrates that a more human-centered analysis can lead to additional insights. We have developed a novel visualization tool designed to assist researchers in improving difficulty detection algorithms. Assuming that data exists from a study in which both predicted programmer difficulties and ground truth were recorded while running an online algorithm for detecting difficulties, the tool allows researchers to interactively travel through a timeline showing the correlation between values of the features used to make predictions, difficulty predictions made by the online algorithm, and ground truth. We used the tool to improve an existing online algorithm based on a study involving the development of a GUI in Java. Episodes of difficulty predicted by the previously developed algorithm were correlated with features extracted from participant logs of interaction with the programming environment and web browser. The visualizations produced from the tool contribute to a better understanding of programmer actions during periods of difficulty, help to identify specific issues with the previous prediction algorithm, and suggest potential solutions to these issues. Thus, the information gained using this novel tool can be used to improve algorithms that help developers receive assistance at appropriate times.

Keywords— Difficulty detection, visualization

I. INTRODUCTION

Previous work suggests that automatic instantaneous detection of programmer difficulty can promote the help given to software developers and students [1], which in turn can increase productivity and learning. Aggregate statistics from previous research efforts have identified promising methods for difficulty detection, but more work is needed in order to fully understand what causes false positive and false negative difficulty predictions. In this paper, we take an alternative approach to analyzing mined programmer data not yet explored in the literature. Using a more human-centered style of analysis aided by a visualization tool that we developed for use by researchers, we examine the correlation between programmer actions and difficulty faced by specific representative programmers in a study.

II. STUDY

In the study, 15 mid- to advanced-level CS students at UNC-Chapel Hill were asked to complete a programming task involving the use of the Java AWT/Swing API. A Firefox plugin was used to track the participants' web history, and the

Fluorite tool [2] extended with our difficulty prediction code was used to gather the participants' programming commands in the Eclipse programming environment. An online algorithm developed by us [1] made predictions of whether the participants were facing difficulty as they were completing the task. Participants were able to correct these difficulty predictions, ask for help, and classify a difficulty as having to do with the high-level design of the solution, not understanding the Java Swing API, or an inability to get the right output. Not all difficulties were classified. More details of the study are given in [3, 4].

III. DATA ANALYSIS AND VISUALIZATION

Our online algorithm divides the raw log of programmer actions into segments and calculates, for each segment of the log, ratios of five classes of user commands: *edit* (i.e. inserting or deleting code), *debug* (i.e. using the debug tool in Eclipse), *focus* (i.e. focusing in and out of Eclipse), *navigation* (i.e. navigating within Eclipse), and *remove-class* (i.e. deleting a class in Eclipse). These command classes are intended to cover the breadth of interactions that occur while programming. The ratios represent the number of commands of a certain class that occurred relative to the total number of commands during that segment. In addition, we calculated the number of web links traversed during a segment - a feature not used in the existing prediction algorithm.

To help understand and refine the correlation between the existing features, web links, and ground truth, we extended a visualization tool we had implemented previously [5]. The extended tool shows the values of programmer command ratios at different times along with the number of web links visited, the predicted difficulty level, the actual difficulty level based on corrections and observations, and the type of difficulty. We used the tool to visualize and analyze the interactions of representative programmers from the study. Figure 1 shows the visualizations for three of these programmers. The green bars represent normal progress and the pink bars represent difficulty points. The code W(number) shows the number of web links traversed during the corresponding segment. The Type bar presents the type of difficulty faced by the programmer, displaying red dots for design, yellow dots for incorrect output, and teal dots for API (Fig. 1, P22). Additional black dots represent insurmountable difficulty (i.e. difficulty accompanied with help requests; Fig. 1, P18).

Our visualizations showed that during the vast majority of segments, programmers did not face difficulty, which should be expected if the task is appropriate for the skill levels of the

subjects. They also showed that some difficulties were detected by the existing algorithm, though there were some false negatives, which is consistent with previous findings [1].

Our visualization-based analysis of individual interactions also revealed information that was not accessible via previous analysis of aggregate statistics provided by Weka. The plots revealed that web accesses and debug commands went up and edit commands went down during periods of difficulty (Fig. 1). However, the command usage patterns surrounding difficulty episodes were programmer independent. Some people faced more difficulties than others: for instance, one participant (P29) had more than ten difficulty episodes, the vast majority of which were correctly predicted by the algorithm, while another (P18) had only three difficulty episodes, none of which were predicted. Interestingly, all false negatives in the plots show web links traversed during the associated segments. Web links were traversed for all difficulty types, not just API-related issues as

we originally anticipated. This indicates that programmers are seeking help online for all types of difficulties faced and suggests that web links could be a useful feature to incorporate in the prediction algorithm in the future. This insight led to a refinement of the algorithm that improved it [3].

In addition, the plots showed that difficulties associated with API, design and incorrect output occurred almost equally and incorrect output difficulty was predicted correctly more often than API or design difficulties. Insurmountable difficulties were shown to be less common than surmountable difficulties. We could not visually see strong correlations between the other features used in the existing algorithm (focus, navigation, and remove-class), which may have to do with the specifics of the API-oriented GUI task.

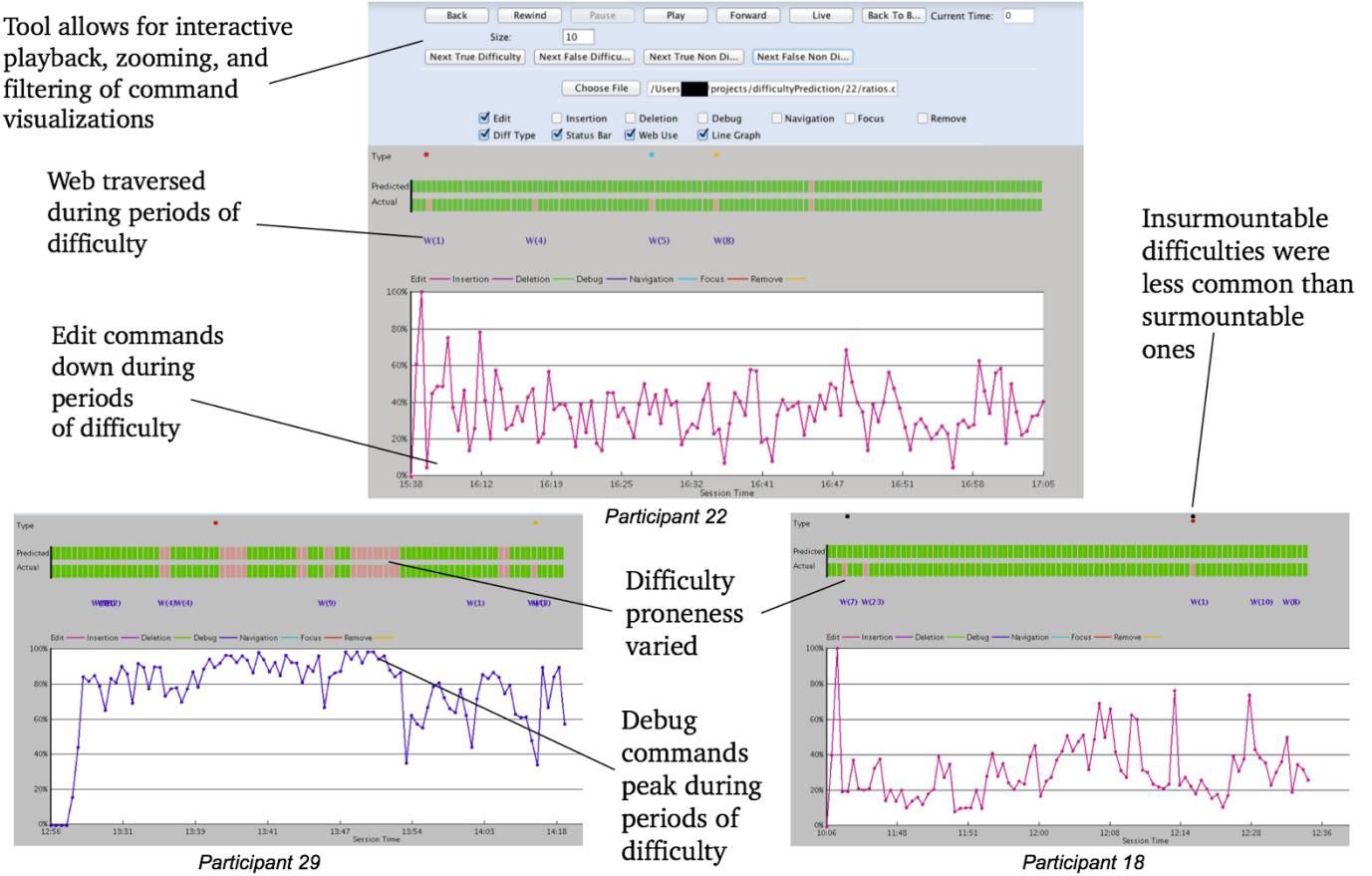


Fig. 1. Interactive visualization of programmer interactions for participants 18, 22, and 29. The line graphs for P22 and P18 are filtered to only show edit command ratios over time; P29's graph shows debug command ratios.

IV. CONCLUSIONS AND FUTURE WORK

Visualizing individual feature-prediction timelines furthers intuitive understanding of the prediction process and is an alternative to the textual aggregate analysis provided by general purpose analysis tools such as Weka [6]. More importantly, it provides a way to pinpoint issues with specific predictions made by an existing algorithm (e.g. the algorithm's failure to predict

specific participants' difficulty episodes) and suggests solutions for improvement (e.g. using web links as a feature). We need to further study these visualizations and analyze the trends we see using quantitative statistics such as information gain. Thus, the information gleaned from this analysis can inform the development of difficulty prediction algorithms that can help developers and students receive assistance at appropriate times.

REFERENCES

- [1] Carter, J. and P. Dewan. Design, Implementation, and Evaluation of an Approach for Determining When Programmers are Having Difficulty. in Proc. Group 2010. 2010. ACM.
- [2] Yoon, Y. and B.A. Myers. Capturing and analyzing low-level events from the code editor. in Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools. 2011. New York.
- [3] Long, D., K. Wang, J. Carter, and P. Dewan. Exploring the Relationship between Programming Difficulty and Web Accesses in Proc. VL/HCC 2018. Lisbon, Portugal: IEEE.
- [4] Carter, J., Automatic Difficulty Detection, in Department of Computer Science. 2014, University of North Carolina (Chapel Hill): Chapel Hill. p. 201.
- [5] Long, D., N. Dillon, K. Wang, J. Carter, and P. Dewan. Interactive Control and Visualization of Difficulty Inferences from User-Interface Commands. in IUI Companion Proceedings. 2015. Atlanta: ACM.
- [6] Witten, I.H. and E. Frank, Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations. 1999: Morgan Kaufmann.

How End Users Express Conditionals in Programming by Demonstration for Mobile Apps

Marissa Radensky
*Computer Science Department
 Amherst College
 Amherst, MA
 mradensky19@amherst.edu*

Toby Jia-Jun Li
*Human-Computer Interaction Institute
 Carnegie Mellon University
 Pittsburgh, PA
 tobyli@cs.cmu.edu*

Brad A. Myers
*Human-Computer Interaction Institute
 Carnegie Mellon University
 Pittsburgh, PA
 bam@cs.cmu.edu*

Abstract—Though conditionals are an integral component of programming, providing an easy means of creating conditionals remains a challenge for programming-by-demonstration (PBD) systems for task automation. We hypothesize that a promising method for implementing conditionals in such systems is to incorporate the use of verbal instructions. Verbal instructions supplied concurrently with demonstrations have been shown to improve the generalizability of PBD. However, the challenge of supporting conditional creation using this multi-modal approach has not been addressed. In this extended abstract, we present our study on understanding how end users describe conditionals in natural language for mobile app tasks. We conducted a formative study of 56 participants asking them to verbally describe conditionals in different settings for 9 sample tasks and to invent conditional tasks. Participant responses were analyzed using open coding and revealed that, in the context of mobile apps, end users often omit desired *else* statements when explaining conditionals, sometimes use ambiguous concepts in expressing conditionals, and often desire to implement complex conditionals. Based on these findings, we discuss the implications for designing a multi-modal PBD interface to support the creation of conditionals.

Keywords—*conditionals, programming by demonstration, verbal instruction, end-user development, natural programming*.

I. INTRODUCTION AND BACKGROUND

Script generalization continues to be the key challenge for programming-by-demonstration (PBD) systems for task automation [1],[2]. A PBD system should not only produce literal record-and-replay macros, but also understand end user intentions behind recordings and be able to perform similar tasks in different contexts [2]. Prior approaches of asking users to provide several examples from which AI algorithms can make generalizations using program synthesis approach, and having users supply the features needed for generalization have been shown to be infeasible due to users' limited ability to understand generalization options and provide sets of useful examples spanning the complete space for synthesizing the intended programming logic. Our research on SUGILITE [3], EPIDOSITE [4], and APPNITE [5] demonstrated that leveraging natural language instructions grounded by mobile apps' GUIs is a promising method to enable users to naturally express their intentions for generalizing PBD scripts. While these systems use natural language instructions to infer script parameterization and data descriptions for individual actions, none address the challenge of enabling users to create task-wide conditionals, an important aspect of generalization.

Evidenced in [6], non-programmers state conditionals using varying structures and levels of description. Understanding the different manners in which end user programmers construct conditionals and whether or not they provide the details necessary for an intelligent agent to comprehend their conditionals is crucial to building a PBD system that can interact with users to extract intended conditionals from verbal instructions. In this extended abstract, we summarize our study on how end users naturally describe conditionals in the context of mobile apps and discuss the implications for designing a multi-modal PBD interface that supports conditionals.

II. METHODS

A. Formative Study

We conducted a formative study on Amazon Mechanical Turk with 56 participants (38 non-programmers; 38 men, 17 women, 1 non-binary person). 30 participants completed a 3-part survey, while 22 completed either Part 1 or 2, both followed by Part 3. The other 4 participants completed both versions of the survey. 11 of 104 utterances in Part 1, 10 of 62 in Part 2, and 19 of 65 in Part 3 were excluded from analysis due to question misunderstandings and blank responses. Each part included an example question and responses.

In Part 1, participants were given a description of a task for an intelligent agent to complete within a PBD system for mobile apps. The task had distinct associated situations, each of which led to the task being completed differently. The participants were assigned one of 9 tasks such as playing a type of music that depends on the time of day or going to a location with a mode of transportation that depends on how much time getting there by public transportation takes. They were asked what they would say to the agent so that it may understand the *difference* among the situations, and then for any alternative responses. To avoid biasing responses' wording, we used the Natural Programming Elicitation method [7], presenting pictures alongside limited text to describe the task and situations.

Part 2 differed in purpose from Part 1 in that it had participants express conditionals while looking at relevant phone screens. Participants were given a mobile app screenshot with yellow arrows pointing to the screen components containing information pertinent to the condition on which the task situation depended. If other components might have been

confused with the correct ones, red arrows pointed them out. Participants were asked to explain to the agent how to locate and use the correct components to determine the situation at hand. Finally, Part 3 asked participants for another task for which an agent should perform differently in distinct situations.

B. Open Coding

The participants' responses were analyzed using open coding. For all 3 parts, a code identified conditionals with unambiguous versus ambiguous language. For Part 1, codes were used to identify conditionals without *else* statements, to categorize the implied necessity of omitted *else* statements, and to identify omitted *else* statements whose contents are implied. For Part 3, codes were used to identify conditionals with complex structures, those that use 2 or more apps, those initiated by automatic triggers, and those with automatic triggers based on information found in open APIs or app GUIs.

III. PRELIMINARY RESULTS AND IMPLICATIONS

A. Omission of Else Statements

In Part 1, though only conditionals with *else* statements were given as example responses, 56% of the 39 participants who completed Part 1 provided at least one response without an *else* statement. Of those participants, 45% omitted an *else* statement even though it was not clear whether it would be needed or not. As an example, “*Whenever I go to bed past 11 p.m. set 3 alarms*” may or may not require an alternative such as setting 1 alarm. Furthermore, 18% omitted an *else* statement when it was definitely necessary. “*Default to upbeat music until 8pm every day*,” for instance, requires an alternative for other times. This finding suggests that end users will often omit the appropriate *else* statement in their natural language instructions. Additionally, merely 33% of participants expressed conditionals that implied the required alternative when it was omitted and possibly or definitely necessary (e.g. “*If a public transportation access point is more than half a mile away, then order an Uber*” implies an alternative of finding a public transportation route). PBD must thus be designed to detect omitted *else* statements in natural language and guide users to resolve ambiguity in conditional alternatives.

B. Ambiguous Concepts in Conditions

6 of the 9 tasks' descriptions deliberately referred to conditions incorporating ambiguous concepts such as “*cold*” and “*daytime*.” To the last 27 participants, only unambiguous example responses were shown to try to guide them away from using ambiguous concepts. 10 of them completed Part 1 for one of the 6 tasks just mentioned. 40% of the 10 participants still supplied an ambiguous condition, such as “*When I am going to outside at chance of rain I will take umbrella ...*” An agent should be able to use multi-turn dialogue to ask users to clarify ambiguous concepts like “*chance of rain*.”

With or without seeing exclusively unambiguous example responses, 25 participants completed Part 2 for one of the 6 potentially ambiguous tasks. Interestingly, in this part in which participants were provided an app screenshot displaying

specific information relevant to their task's condition, all 25 participants provided clear definitions such as “*longer than an hour*” and “*past 8:00 pm*” for ambiguous concepts. However, 15 participants who were given all unambiguous example responses completed Part 3, in which participants invented their own conditional tasks, and 20% of these 15 participants expressed conditions that contained ambiguous concepts. These results suggest that users might eliminate ambiguity from their conditions by describing them while looking at the relevant mobile app GUIs. If users still use ambiguous concepts, they may be guided to disambiguate their conditions by prompts to explain the ambiguous concepts in the context of the GUIs.

C. Desired Conditionals

Many participants desired conditionals that were complex in some manner. 55% of the 44 invented conditionals use more than 1 app, and 9% use more than 2 apps. 14% of the conditionals, such as the switch statement “*if it is day X, order food Y*,” have a more complicated structure than just “*If ... else ...*.” Also, automatic triggers instead of voice commands must initiate 55% of the conditionals, and 58% of these triggers are not simple triggers like a notification but rather information found in open APIs or app GUIs. For instance, “*Turn the light on in the room if I'm at home at sunset or when I arrive home after sunset*” has a trigger involving the user's location, time of sunset, and current time, all information in open APIs. These results motivate our PBD system, which allows users to develop scripts for cross-app tasks more complex and personalized than common pre-programmed ones.

We are now researching how to augment SUGILITE [3] and APPINITE [5] to have all the indicated functionalities.

ACKNOWLEDGMENT

This research was supported in part by Oath through the InMind project and in part by NSF grants CCF-1560137 and CCF-1814826. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsors.

REFERENCES

- [1] H. Lieberman, *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.
- [2] A. Cypher and D. C. Halbert, *Watch what I do: programming by demonstration*. MIT press, 1993.
- [3] T. J.-J. Li, A. Azaria, and B. A. Myers, “SUGILITE: Creating Multimodal Smartphone Automation by Demonstration,” in *Proceedings of CHI 2017*.
- [4] T. J.-J. Li, Y. Li, F. Chen, and B. A. Myers, “Programming IoT Devices by Demonstration Using Mobile Apps,” in *Proceedings of IS-EUD 2017*.
- [5] T. J.-J. Li *et al.*, “APPINITE: A Multi-Modal Interface for Specifying Data Descriptions in Programming by Demonstration Using Natural Language Instructions,” in *Proceedings of VL/HCC 2018*.
- [6] J. F. Pane, B. A. Myers, and others, “Studying the language and structure in non-programmers' solutions to programming problems,” *Int. J. Hum.-Comput. Stud.*, vol. 54, no. 2, pp. 237–264, 2001.
- [7] Brad A. Myers, Andrew J. Ko, Thomas D. LaToza and YoungSeok Yoon. “Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools,” *IEEE Computer*. 2016. vol. 49, no. 7. pp. 44-52.

Educational Impact of Syntax Directed Translation Visualization, a Preliminary Study

Damián Nicolalde-Rodríguez

*School of Engineering**Pontificia Universidad Católica del Ecuador*

Quito, Ecuador

danicolalde@puce.edu.ec

Jaime Urquiza-Fuentes

*LITE - Laboratory of Information Technology and Education**Universidad Rey Juan Carlos*

Madrid, Spain

jaime.urquiza@urjc.es

Abstract—This work studies the effect of using software visualization to teach syntax directed translation, a complex topic within compiler subjects. A trial was conducted with 34 students using LISA as the visualization tool. It was divided in two phases. Firstly, student's experience during compilers construction labs was studied, comparing LISA versus CUP. All participants used both tools and answered a questionnaire. LISA was scored as more motivational and easier to use. Moreover, key theoretical concepts were better identified with LISA. Secondly, a typical lecture (control group) was compared against a lecture using LISA (treatment group). Students were randomly distributed between both groups and answered a knowledge test following the lectures. Results showed that the treatment group significantly outperformed the control group. However, areas for improvement have been detected even in the treatment group. These improvements could be addressed by enhancing the visualization tool with features to increase student engagement.

Index Terms—Software visualization, Compilers, Educational technologies

I. INTRODUCTION

Visualization has been being used by humans to gain understanding of complex problems for centuries. Educational use of visualization based technology is neither new. Back in the eighties Baecker & Sherman generated one of the first algorithm visualizations entitled “Sorting out Sorting” [1]. Since the beginning of this field, many teachers felt that visualization could be a effective educational tool, being used in many subjects.

Language Processors and compilers are among the most complex subjects within CS degrees. And visualization based educational tools can be found as well, but most of them deal with automata theory, lexical analysis and parsing, e.g. JFlap [2]. This work is focused on Syntax Directed Translation (SDT), a complex topic of these subjects where few tools can be found and less evaluations have been published.

In this poster we present a preliminary evaluation of the impact of the use of visualizations dedicated to SDT. Firstly, the use of parser generators with and without visualization features is studied. Secondly, the differences between receiving class in a traditional way and using a visualization tool are analyzed in terms of student learning. Both studies have used LISA[3] as the TDS visualization tool. They have been

carried out during the Compilers and Interpreters course at the Universidad Pontificia Universidad Católica del Ecuador, in the Systems and Computer Engineering Degree during two academic years (2016-2017 and 2017-2018). The number of participants was 34: 12 the first year and 22 the second. All students passed the basic programming, data structure, object-oriented programming, language design and automata courses and had prior knowledge of Lexical Analysis, Parsing and SDT.

II. PARSER GENERATORS WITH/WITHOUT VISUALIZATION

This study analyzes students' perception regarding the use of software tools that generate visualizations, versus those that do not. Non-visualization tools were represented by CUP (<http://www2.cs.tum.edu/projects/cup/>), an LALR parser generator quite similar to well known ones, e.g. YACC or BISON.

In order to perform a comparative analysis all students used both tools. The study lasted two hours and consisted of four phases. Firstly, the teacher gave a brief review about the theoretical concepts to be used during the session. Secondly, the LISA tool was used by the students to generate an SDT. The SDT specification was provided by the teacher. Thanks to the visualization features of LISA he gave a visual explanation about the construction process of the syntax tree, the execution of semantic actions and the evaluation and communication of attribute values. Subsequently, students could animate the evaluation tree (syntax tree annotated with attribute values) trying their own inputs. Thirdly, Cup was used to generate an SDT with the same requirements as the one used before. Again, the teacher provided the students with the specifications and the students tried the SDT generated by CUP with their own inputs. The main difference was that students could view how all tokens were processed by the lexical analyzer but only the final result of the SDT execution. Finally, the students completed a questionnaire about their experience using each tool. This questionnaire had four parts: theoretical concepts identification; teaching-learning methodology; ease of use, installation and configuration; and student-software interaction.

Students' answers suggest that both tools allowed them to recognize the three basic phases of a language processor but none of the tools allowed them to identify the underlying

parsing algorithm. Only LISA allowed students to recognize the kind of attributes used in the SDT specification, but this was an expected result because CUP only provides the final result of the SDT execution. Regarding teaching-learning methodology, more than 94% of the students thought that LISA was the tool that achieves the greatest motivation in the teaching-learning process. In addition, 76.5% of students said that LISA is intuitive and user friendly while 91% said the opposite about CUP. Finally, more than 91% of the students think that LISA allows the student to interact with the phases of the language processor in a dynamic way while CUP does not.

III. CLASSROOM: VISUALIZATION SW VS. BLACKBOARD

The main objective of this study is to verify if the use of LISA, the SDT visualization tool improves students' performances when compared against a traditional class where blackboard and markers are used. In this study students were randomly divided into two equal groups (17 students each group): control group and treatment group. The instructor was the same for both groups. The session lasted 60 minutes, begun with an introductory explanation of the SDT concepts and ended with a set of problems for both groups.

The control group received the SDT class in a traditional way through a master class where the teacher based his explanations on visualizations (graphs) made on the blackboard with the support of slides. Thus, the students observed these explanations and graphs drawn by the teacher on the blackboard taking notes in their notebooks. The teacher also asked students to solve problems in their notebooks. These problems consisted in simulating and explaining the behavior of the specification.

The treatment group received the same class, but the teacher gave explanations with the support of LISA, using the animations generated by the tool and asking the students to carry out the problems using the tool.

In order to test if there is any difference between both teaching methods a knowledge questionnaire was provided to the students. The instrument was applied to the students individually. Question one sought to determine the ability acquired by the student to identify how the tokens were identified by lexical rules, the definition of the attributes and the semantic rules. The second, third and fourth questions analyzed the student's ability to understand the lexical rules, and based on these, the input chain supported. The fifth question determined whether the student was able to differentiate the synthesized and/or inherited attributes in the specification. The sixth question helped to determine if the student understood how parser requests a new token from the lexical analyzer to build the syntax tree from the input string. The seventh and eighth questions sought to determine whether the student understood the SDT concepts and can identify the evaluation of the attributes in the annotated parsing tree, and how it interacts with the specification.

Considering the whole knowledge questionnaire (in a 0-10 scale), the treatment group ($M=7.41, SD=1.62$) significantly

outperformed the control group ($M=4.03, SD=2.26$), $p=1.85e-05$. This result is also supported by an effect size analysis with Cohen's $d=1.71$.

Analyzing each question (using % of successful answers per group and p-value) it can be seen that the students in the treatment group identify the lexical rules, the definition of the attributes and the semantic rules better than those of the control group (treatment=94.12%, control=62.75%, $p=0.0048$). The results of the sixth question indicate that students in the treatment group better understand the construction of the syntax tree from the input string (treatment=87.25%, control=58.04%, $p=0.0055$). Based on the results of the seventh question it can be said that the treatment group evaluate the attributes in the annotated syntax tree better than the control group (treatment=88.82%, control=54.71%, $p=0.0031$). The results of the eighth question show that the students of the treatment group better understand the interaction between the specification and the syntax analysis tree when attributes are evaluated at a given time (treatment=54.25%, control=22.61%, $p=0.0008$). But in this aspect students' understanding could still be improved. The students of both groups equally understand concepts asked in the rest of the questions.

IV. CONCLUSIONS

Taking into account the results of this preliminary study, we think that visualization could be an effective learning tool. The first study has shown that the use of LISA, the visualization tool, motivates students to participate actively during class because it supports a significant connection between theory and practice concepts. In addition, most of students feel more comfortable when they use LISA. Results obtained with CUP, the non-visualization tool, were clearly worse than those obtained with LISA. Results of the second study indicate that there is a significant improvement in student's performance when the class is taught using LISA instead of a classical approach with blackboard and markers.

We have also detected some improvements regarding two aspects: the visualization of the underlying parsing algorithm (LR or LL) and the interactive features of the visualizations provided by LISA. These results will guide our future efforts.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the Spanish Ministry of Economy and Competitiveness [grant number TIN2015-66731-C2-1-R]

REFERENCES

- [1] R. Baecker and D. Sherman, "Sorting out sorting," 1981. [Online]. Available: <https://www.youtube.com/watch?v=SJwEwA5gOkM>
- [2] S. Rodger, J. Genkins, I. McMahon, and P. Li, "Increasing the experimentation of theoretical computer science with new features in jflap," in *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '13. New York, NY, USA: ACM, 2013, pp. 351–351. [Online]. Available: <http://doi.acm.org/10.1145/2462476.2466521>
- [3] M. Mernik, M. Lenic, E. Avdicusevic, and V. Zumer, "Compiler/interpreter generator system lisa," in *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, Jan 2000, pp. 1.1–1.10.

Semantic Clone Detection: Can Source Code Comments Help?

Akash Ghosh

Tandy School of Computer Science

University of Tulsa

akashghosh@utulsa.edu

Sandeep Kaur Kuttal

Tandy School of Computer Science

University of Tulsa

sandeep-kuttal@utulsa.edu

Abstract—Programmers reuse code to increase their productivity, which leads to large fragments of duplicate or near-duplicate code in the code base. The current code clone detection techniques for finding semantic clones utilize Program Dependency Graphs (PDG), which are expensive and resource-intensive. PDG and other clone detection techniques utilize code and have completely ignored the comments - due to ambiguity of English language, but in terms of program comprehension, comments carry the important domain knowledge. We empirically evaluated the accuracy of detecting clones with both code and comments on a JHotDraw package. Results show that detecting code clones in the presence of comments, Latent Dirichlet Allocation (LDA), gave 84% precision and 94% recall, while in the presence of a PDG, using GRAPLE, we got 55% precision and 29% recall. These results indicate that comments can be used to find semantic clones. We recommend utilizing comments with LDA to find clones at the file level and code with PDG for finding clones at the function level. These findings necessitate a need to reexamine the assumptions regarding semantic clone detection techniques.

I. INTRODUCTION

“Don’t reinvent the wheel, just realign it.” A common practice for programmers to increase their productivity is copying an existing piece of code and changing it to suit a new context or problem. This reuse mechanism promotes large fragments of duplicate or near-duplicate code in the code base [2]. These duplicates are called code clones. Research shows that about 7% to 23% of software systems contain duplicated codes [9]–[12].

In software engineering, many techniques [1] have been proposed to detect code clones based on token similarity (e.g., CCFinder [18], CloneMiner [19] and CloneDetective [17]), Abstract Syntax Tree(e.g., CloneDR [13], Deckard [14]) and Program Dependency Graph (e.g., [3], [6], [7], [15], [16]). One of the most challenging types of clones to find are semantic clones - code fragments that are functionally similar but may be syntactically different. Techniques based on Program Dependency Graph (PDG) are one of the most notable mechanisms to detect semantic code clones [3] as it abstracts many arbitrary syntactic decisions that a programmer made while constructing a function. However, PDG-based techniques are computationally expensive, as they require resource-intensive operations to detect the clones.

Current code clone detection techniques do not include source comments. From a program comprehension point of

view, these comments carry important domain knowledge and also might assist other programmers to understand the code. One of the reasons, to ignore code comments is due to the ambiguity of the English language. For humans, it is easy to comprehend the similarity or difference between words or topics, but a machine may treat the words differently. However, with recent advancement in machine learning and natural language processing tools we hope to detect clone sets by using LDA.

In this paper, we investigated:

- **RQ1:** Does the use of comments help in detecting semantic clones in the code base?
- **RQ2:** Does a PDG based technique, which just uses code for detection of semantic clones, perform equivalently to an LDA based technique, which uses comments?

II. METHODOLOGY

A. Dataset

In this work, JHotDraw-a java package-has been used which contains 310 java source files with 27kLOC. JHotDraw [8] has been widely used in clone detection studies [5].

B. Procedure

1) **PDG:** GRAPLE [3], [4], an existing PDG based clone detection tool, was used to identify clones within the Java package and JPDG to create an undirected graph (vertex-edge, veg) for the whole package. The tool generates a JSON file with edges and vertices in the form of a dictionary. This veg file was then used as an argument along with min-support, sample-size, min-vertices, and selection probability for GRAPLE. The clone sets were generated with and without the selection probabilities with support=5, sample-size=100, and min-vertices=8.

2) **LDA:** Python 3.6 and Regex Expression were used to extract the comments from the source files. All comments were included, except the copyright comments, since it does not contain any information related to the functionality of the source code. Once the comments were extracted, it was normalized by cleaning the stop words and the punctuations. With this normalized texts, a dictionary was created which was used to create the Doc-Term matrix. The LDA model was trained using the corpus and dictionary mentioned above. Then the passes and iterations were set to a specified value. The

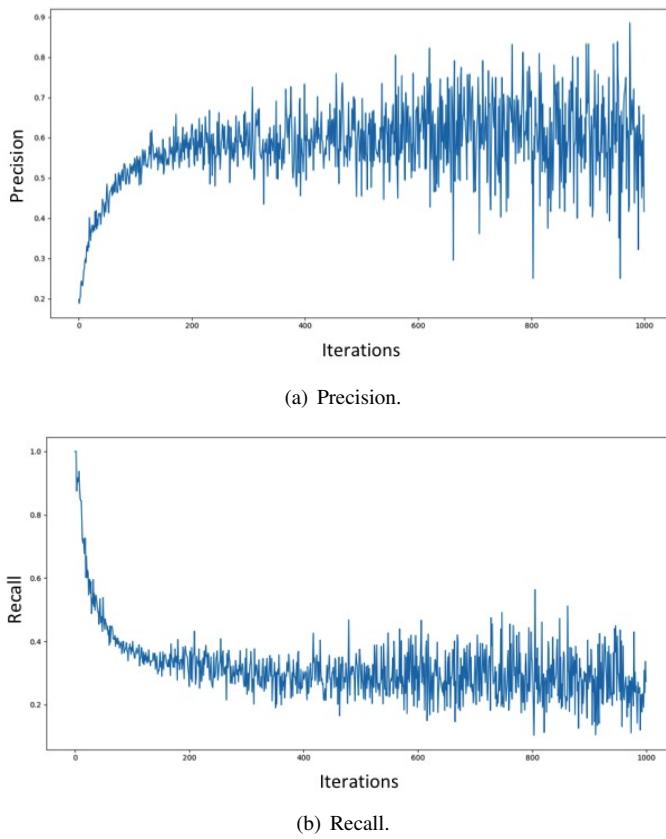


Fig. 1. Precision and Recall.

comment files were passed as an argument to the model to generate the relative topics. Once all the clone sets were generated, we calculated the precision - $|D_{reported} \cap D_{actual}| \div |D_{reported}|$ and recall - $|D_{reported} \cap D_{actual}| \div |D_{actual}|$.

$D_{reported}$ is the set of multi-sets reported by the model and D_{actual} is the ground truth which contains 52 clone sets built manually in 45 hours.

III. RESULTS

A. RQ1: Can code comment help?

To understand whether comments can assist in detecting code clones, the model was trained and the outputs (clone sets) were analyzed in two different ways.

1) *Way 1*: The LDA model was trained using the files as the corpus. With topic limit set to 100, we were able to extract 66 clone sets (274 files). The precision and recall found are mentioned in Table 1.

2) *Way 2*: To understand how the clone sets varied in terms of precision and recall, the model was trained over a range of 1 to 1000 topics. The parameters were set at 1000 iterations with 50 passes. Table 1 shows the best precision found with topic set to 975, which generated 7 clone sets with 21 files. From Fig.1 it is evident that with increased iterations fewer clone sets were found. Also, as the number of iterations increased the precision increased as well with a global maxima at 975. However, the recall decreased.

To further add, the best clone sets in comparison with the ground truth were the clones sets generated by topic number 975. The clone sets were manually analyzed to check the authenticity, it was observed that the matched clone sets i.e $|D_{reported} \cap D_{actual}|$ have high similarities in terms of object or instance creation.

B. RQ2: PDG vs LDA: code vs comments?

Further, to compare a PDG based technique with LDA, we used GRAPLE [3]. We evaluated GRAPLE with and without the selection probability P_i , the later was used to avoid the “Curse of Dimensionality”.

1) *Without Pr*: In this evaluation technique the sample-size were varied multiple times, setting it from 20 to 200, but in most of the cases very small increase in clone sets were observed. Precision and recall mostly varied between 50% to 55%. Table 1 depicts the precision and recall for sample-size 100 with min-vertices 8 and support set to 5.

2) *With Pr*: Using the selection probabilities and with the above mentioned specification, we generated 80 clone sets. Table 1 shows the 22 clone sets were found while using selection probabilities, and 17 clone sets were found without using selection probabilities. Comprehensive clone sets were reported by the model with probability in expense of 30 hours and 74 GB of memory. However, the model without probability reported 17 clone sets in 4.5 secs and consumed 481.5 MB. Moreover, 16 out of 17 clone sets which were reported by model without probability were also reported by the model with probability.

Evidently, the precision and recall for LDA are better than PDG. Upon analyzing the clone sets returned by PDG and LDA, it was observed that LDA was able to find more clone sets. In addition, LDA quickly found the clones based on similar comments compared to PDG, which took hours. Our dataset consists of 27 KLOC, so for such packages PDG based techniques can perform decently, but for larger sizes as noticed by [3] they can deplete the resources.

TABLE I
PRECISION AND RECALL FOR LDA AND PDG.

		#clonesets	Recall	Precision
LDA	Way 1	66	94.86	84.21
	Way 2	7	28.61	88.57
PDG	Without Pr.	17	27.84	52.94
	With Pr.	22	28.7	55.39

IV. CONCLUSION

Our results show that comments can be utilized with LDA and are equivalent to sophisticated PDG based techniques. One approach would be using comments with LDA to detect clone sets at the file level, as this process is less resource-intensive, and applying PDG based code detection techniques at the function level. Our study provides the very first evidence that comments which are underrated in clone detection research can be utilized effectively.

REFERENCES

- [1] C.K. Roy, M.F. Zibran, and R. Koschke, “The vision of software clone management: Past, present, and future (Keynote paper)”, in Proceedings of Software Maintenance, Re-engineering and Reverse Engineering, pp.18-33, 2014.
- [2] J. Howard Johnson, “Visualizing textual redundancy in legacy source”, in Proceedings of Centre for Advanced Studies on Collaborative, pp.32, 1994.
- [3] TAD Henderson and A Podgurski, “Sampling code clones from program dependence graphs with GRAPLE”, in Proceedings of International Workshop on Software Analytics, pg 47-53, 2016.
- [4] H. Cheng, X. Yan, and J. Han, “Mining Graph Patterns. In Frequent Pattern Mining”, in Managing and Mining Graph Data, pp.307-338, 2010.
- [5] Y. Lin, Z. Xing, Y. Xue, Y. Liu, X. Peng, J. Sun, and W. Zhao, “Detecting differences across multiple instances of code clones”, in Proceedings of International Conference on Software Engineering, pp.164-174, 2014.
- [6] J. Krinke, “Identifying similar code with program dependence graphs”, in Proceedings of Working Conference on Reverse Engineering, pp.301-309, 2001.
- [7] R. Komondoor and S. Horwitz, “Using Slicing to Identify Duplication in Source Code”, in Proceedings of International Symposium on Static Analysis, pp.40-56, 2001.
- [8] JHotDraw: <http://www.jhotdraw.org/>
- [9] B. Baker, “On Finding Duplication and Near-Duplication in Large Software Systems”, in Proceedings of Working Conference on Reverse Engineering, pp.86-95, 1995.
- [10] I. Baxter, A. Yahin, L. Moura and M. Anna, “Clone Detection Using Abstract Syntax Trees”, in Proceedings of International Conference on Software Maintenance, pp.368-377, 1998.
- [11] C. Kapser and M. Godfrey, “Supporting the Analysis of Clones in Software Systems: A Case Study”, Journal of Software Maintenance and Evolution: Research and Practice - IEEE International Conference on Software Maintenance, Vol.18 (2), pp.61-82, 2006.
- [12] J. Mayrand, C. Leblanc and E. Merlo, “Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics”, in Proceedings of Proceedings of International Conference on Software Maintenance, pp.244-253, 1996.
- [13] F. Al-Omari, I. Keivanloo, C. K. Roy, and J. Rilling, “Detecting clones across microsoft .net programming languages”, in Proceedings of Working Conference on Reverse Engineering, pp.405-414, 2012.
- [14] S. Bazrafshan, and R. Koschke, “An empirical study of clone removals”, in Proceedings of International Conference Software Maintenance, pp.50-59, 2013.
- [15] D. Chatterji, J. C. Carver, and N. A. Kraft, “Cloning: The need to understand developer intent”, in International Workshop on Software Clones, pp. 14-15, 2013.
- [16] J. R. Cordy, “Comprehending reality: Practical barriers to industrial adoption of software maintenance automation”, in International Workshop on Program Comprehension, pp.196-206, 2003.
- [17] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. Hassan, “An empirical study on inconsistent changes to code clones at the release level”, in Working Conference on Reverse Engineering, pp.760-776, 2012.
- [18] S. Bouktif, G. Antoniol, M. Neteler, and E. Merlo, “A novel approach to optimize clone refactoring activity”, in Proceedings of Conference on Genetic and Evolutionary Computation, pp.1885-1892, 2006.
- [19] E. Adar and M. Kim, “SoftGUESS: Visualization and exploration of code clones in context”, in Proceedings of International Conference on Software Engineering, pp.762-766, 2007.

What Makes a Good Developer? An Empirical Study of Developers' Technical and Social Competencies

Cheng Zhou

Tandy School of Computer Science
University of Tulsa
Tulsa, Oklahoma
cheng-zhou-73@utulsa.edu

Sandeep Kaur Kuttal

Tandy School of Computer Science
University of Tulsa
Tulsa, Oklahoma
sandeep-kuttal@utulsa.edu

Iftekhar Ahmed

School of Elect. Eng. & Computer Science
Oregon State University
Corvallis, Oregon
ahmedi@oregonstate.edu

Abstract—Technical and social competencies are highly desirable for a protean developer. Managers make hiring decisions based on developer's contributions to online peer production sites like GitHub and Stack Overflow. These sites provide ample history regarding developers' technical and social skills. Although these histories are utilized by hiring tools to help managers make their hiring decisions, little is known empirically how developers' social skills affect their technical skills and vice versa. Without such knowledge, tools, research, and training might be flawed.

We present an in-depth empirical study investigating the correlation between the technical and social skills of developers. Our quantitative analysis of factors influencing the social skills of developers compared with factors affecting their technical skills indicates that better collaboration competency skills are associated with enhanced coding abilities as well as the quality of code.

I. INTRODUCTION

Technical and social competencies are vital for a successful developer. Developers are using online peer production sites like GitHub for software development and Stack Overflow for learning, which provide ample histories regarding developers' social and technical activities. These are used as a proxy for measuring their social and technical competencies [3], [4], [8]. Managers are using these proxies to assess a potential candidates for hiring in their teams or companies [4]–[9].

Technical Skills are vital for writing code. The two most important skills revealed in the literature are coding competency and quality of work. *Coding ability*: How proficient is an individual's knowledge and ability to code? On a global platform, these skills can be measured by the log activities, number of projects owned or forked, number and frequency of commits/issues/comments and number of languages the professional is proficient in [1], [7]. *Quality of work*: How good is the code that an individual produces? It can be measured by number of accepted commits and inclusion of test cases [5], [6], [9], [10].

Social Skills are soft skills that measure the ability to work as an individual and in teams. Three important skills are collaboration proficiency, project management ability, and

978-1-5386-4235-1/18/\$31.00 © 2018 IEEE

motivation. *Collaboration proficiency*: How well can an individual work with other team members? This is measured by communication activity through the number of comments/answers/questions and reputation. Good team players are vital for the success and timely release of large projects [2]. *Project management ability*: How well can an individual manage the project? This can be measured by the number of projects owned by an individual [6]. *Motivation*: How passionate is an individual about the project? These can be measured by number of the commits/issues/comments of the contributions, non-related side projects, and diversity of languages known [9].

There exists little knowledge about which of the technical or social skills are important and what correlation exists between them. This is the basis of our study.

II. METHODOLOGY

We used data dumps, GHTorrent [11]–[13] and Stack Exchange [14]. To find common active users, we selected users who provide their GitHub link on their Stack Overflow profiles, and we filtered out those who were not active contributors on GitHub by established criterias [15]. First, we removed the projects from our analysis that didn't have language information in the GHTorrent database. Declaring the languages used in a project is a part of the initial setup of a project in GitHub, and missing information in this field raises concerns about the validity of the project. In order to make sure that our results are free from such noise, we filtered out those projects. Secondly, to avoid personal projects, we set a standard that projects should have at least five committers. We found 467,770 GitHub projects from 12,831 common users (on GitHub and Stack Overflow), and after implementing criterias, we were left with 3,266 projects and 1,749 users. We retrieved the data from Stack Exchange for all 1,749 users in Stack Overflow and had 221,219 comments, 19,635 questions, and 90,795 answers.

III. RESULTS

The goal of this paper is to investigate which technical skills or social skills are important when it comes to measuring

TABLE I: Linear model with coefficients

	Dependent Variable	Quality Inferred	# of Answers	Reputation score	# of Questions	# of Contributed projects	McFadden Pseudo R-squared value
RQ1	# of Project owned	Coding ability	3.93e-04	-8.44e-07	1.25e-03	1.34e-01	0.13
	# of Commits	Coding ability	7.78e-04	-2.57e-06	8.72e-04	1.20e-01	0.26
	# of Issues	Coding ability	2.30e-03	-2.95e-06	2.30e-03	8.08e-02	0.27
	# of Comments	Coding ability	1.12e-03	3.46e-06	-1.62e-03	1.50e-01	0.20
	Languages used	Coding ability	8.68e-04	-1.52e-06	null	5.61e-02	0.01
RQ2	# of Accepted commits	Quality of work	1.28e-03	1.52e-06	-2.95e-03	1.49e-01	0.17
	Test case inclusion	Quality of work	3.59e-03	-1.52e-05	3.61e-03	1.36e-01	0.12

the competency of a developer. To answer this overarching question, we analyzed the correlation between various competency measures, and built models using various factors to understand how effective these factors are in explaining technical and social skills. Hence, we targeted two research questions presented here.

RQ1: Which is the most important factor among social skills in relation to Coding ability - a technical skill?

We attempted to identify whether *motivation*, *project management ability*, or *collaboration proficiency* was the most effective factor for coding ability. In order to answer this question, we first computed the pearson correlation coefficients for all the factors. As visible in Figure 1, none of the factors are highly associated with each other.

Next, we built Poisson regression models using all of the coding ability indicators, such as # of Project owned, # of commits etc. with a log linking function and filtered the factors with $VIF > 5$. The significant contributors towards individual's *coding ability* are shown in RQ1 section of Table I. The McFadden Pseudo R-squared [16] for the models are shown in RQ1 section of Table I. We used McFadden's Pseudo R-squared as a quality indicator of the model because there is no direct equivalent of R-squared for Poisson regression. The ordinary least square (OLS) regression approach to goodness-of-fit does not apply for Poisson regression. Moreover, pseudo R-squared values like McFadden's cannot be interpreted as one would interpret OLS R-squared values. McFadden's Pseudo R-squared values tend to be considerably lower than those of the R-squared and values of 0.2 to 0.4 represent an excellent fit.

Next, we wanted to check the kind of quality inferred (discussed in the introduction) by these factors. From RQ1 section of Table II, we can see that factors associated with *collaboration proficiency* are most frequently identified as significant when we try to build models to predict coding abilities of a contributor.

RQ2: Which is the most important factor among social skills for Quality of work - a technical skill?

Our second research question attempted to identify whether *motivation*, *project management ability*, or *collaboration proficiency* was the most important factor in determining the *quality of work*. We followed the same procedure of building Poisson regression models using all of the *quality of work* indicators with a log linking function, shown in Table I.

Then we looked into the category of the factors based on the quality inferred to discover the most frequent category as shown in Table II. *Collaboration proficiency* is the most common factors that is associated with the quality of work.

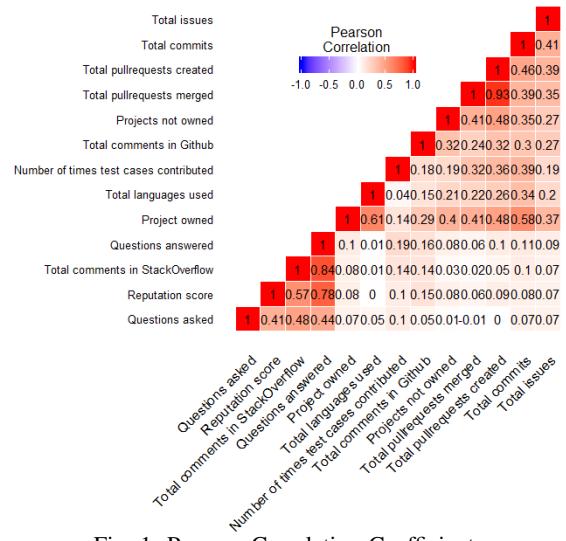


Fig. 1: Pearson Correlation Coefficients

TABLE II: Factor wise frequency along with their categories

	Factors	Freq.	Quality Inferred
RQ1	# of Answers	5	Collaboration proficiency
	Reputation score	5	Collaboration proficiency
	# of Contributed projects	6	Motivation
	# of Questions	4	Collaboration proficiency
	# of Languages	3	Motivation
	# of Projects owned	2	Project management ability
	# of Accepted commits	2	Quality of work
RQ2	# of Answers	2	Collaboration proficiency
	Reputation score	2	Collaboration proficiency
	# of Contributed Projects	2	Motivation
	# of Questions	2	Collaboration proficiency
	# of Languages	2	Motivation
	# of Projects owned	2	Project management ability
	# of Comments	2	Collaboration proficiency

IV. CONCLUSION

In our large scale study, we find that collaboration proficiency is the most frequently identified competency category, and there is a lack of strong association between technical and social skills. The results reaffirm that collaboration is an important factor while developing large software, but there is a lack of strong association between technical and social competency. This opens up an opportunity to identify the reason behind such lack of association and also instigates the need for longitudinal studies to investigate the association over time.

REFERENCES

- [1] Al-Ani, Ban, Matthew J. Bietz, Yi Wang, Erik Trainer, Benjamin Koehne, Sabrina Marczak, David Redmiles, and Rafael Prikladnicki. "Globally distributed system developers: their trust expectations and processes." In conference on Computer supported cooperative work. ACM, 2013.
- [2] Al-Ani, Ban, and David Redmiles. "In strangers we trust? Findings of an empirical study of distributed teams." In International Conference on Global Software Engineering. IEEE, 2009.
- [3] Kristof-Brown, Amy, Murray R. Barrick, and Melinda Franke. "Applicant impression management: Dispositional influences and consequences for recruiter perceptions of fit and similarity." In Journal of Management 28.1 (2002): 27-46.
- [4] Long, Ju. "Open Source Software Development Experiences on the Students' Resumes: Do They Count?-Insights from the Employers' Perspectives." In Journal of Information Technology Education: Research 8 (2009): 229-242.
- [5] Movshovitz-Attias, Dana, et al. "Analysis of the reputation system and user contributions on a question answering website: Stackoverflow." In IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining. ACM, 2013.
- [6] Marlow, Jennifer, and Laura Dabbish. "Activity traces and signals in software developer recruitment and hiring." In conference on Computer supported cooperative work. ACM, 2013.
- [7] Marlow, Jennifer, Laura Dabbish, and Jim Herbsleb. "Impression formation in online peer production: activity traces and personal profiles in github." In conference on Computer supported cooperative work. ACM, 2013.
- [8] Sarma, Anita, et al. "Hiring in the global stage: Profiles of online contributions." In Conference on Global Software Engineering. IEEE, 2016.
- [9] Singer, Leif, et al. "Mutual assessment in the social programmer ecosystem: an empirical investigation of developer profile aggregators." In conference on Computer supported cooperative work. ACM, 2013.
- [10] Tsay, Jason, Laura Dabbish, and James Herbsleb. "Influence of social and technical factors for evaluating contribution in GitHub." In international conference on Software engineering. ACM, 2014.
- [11] Ghtorrent, <http://ghtorrent.org>, accessed: Nov 2016.
- [12] Gousios, Georgios, and Diomidis Spinellis. "GHTorrent: GitHub's data from a firehose." In conference on Mining software repositories. IEEE, 2012.
- [13] Gousios, Georgios. "The GHTorrent dataset and tool suite." In conference on mining software repositories. IEEE, 2013.
- [14] Stack exchange data explorer, <https://data.stackexchange.com>, accessed: Feb 2018.
- [15] Kalliamvakou, Eirini, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. "The promises and perils of mining GitHub." In conference on mining software repositories. ACM, 2014.
- [16] Hensher, David A., and Peter R. Stopher, eds. Behavioural travel modelling. Taylor & Francis, 1979.

Visualizing Path Exploration to Assist Problem Diagnosis for Structural Test Generation

Jiayi Cao¹, Angello Astorga¹, Siwakorn Srisakaokul¹, Zhengkai Wu¹, Xueqing Liu¹, Xusheng Xiao², Tao Xie¹

¹University of Illinois at Urbana-Champaign, ²Case Western Reserve University

Email: {jcao7,aastorg2,srisaka2,zw3,xliu93,taoxie}@illinois.edu, xusheng.xiao@case.edu

Abstract—Dynamic Symbolic Execution (DSE) is among the most effective techniques for structural test generation, i.e., test generation to achieve high structural coverage. Despite its recent success, DSE still suffers from various problems such as the boundary problem when applied on various programs in practice. To assist problem diagnosis for structural test generation, in this paper, we propose a visualization approach named PexViz. Our approach helps the tool users better understand and diagnose the encountered problems by reducing the large search space for problem root causes by aggregating information gathered through DSE exploration.

I. INTRODUCTION

Dynamic Symbolic Execution (DSE) [1]–[3] is among the most effective techniques for structural test generation [4]. DSE collects the constraints on inputs from executed branches to form path conditions and flips constraints in the path conditions to obtain new path conditions for exploring new paths and achieving high structural coverage. However, users of DSE-based tools such as Pex [5]–[7] often experience various categories of problems [8]–[11] while applying the tools on various programs in practice. One major category of problems is the boundary problem: when covering a branch in the program under test requires a large number of explored paths, DSE may not be able to cover such branch due to insufficiency of its default exploration-resource allocation (such as the maximum number of explored paths allocated to path exploration). Such problem often occurs for a program under test containing loops [12] or complex string operations [13].

When such problem arises, the DSE-based tools present little information about the problem root cause, leaving the users in the dark. Furthermore, there is little visual (i.e., easy to digest) guidance readily available to solve the problem effectively. The lack of guidance is especially troublesome given that the tools do not scale well when the number of problems increases. As a result, the time needed to investigate the problem root cause is prohibitive.

To address such issue, in this paper, we propose a visualization approach named PexViz. Our approach helps the tool users better understand and diagnose the encountered problems by reducing the large search space for problem root causes by aggregating information gathered through DSE exploration. In particular, our approach provides visualization to summarize the path-exploration results by collapsing redundant exploration results through a Variant Control Flow

Graph (VCFG) (a CFG with its nodes being reduced to only those corresponding to branch statements) and then encoding information gathered from the DSE process on top of the VCFG. By iteratively interacting with the resulting graph, the users of a DSE-based tool can navigate through relevant information when diagnosing the encountered problems. We implement our approach as an extension to IntelliTest (derived from Pex [5]–[7]), an industrial test generator available in Visual Studio 2015/2017, and a significant improvement over an existing state-of-the-art visualization approach, SEViz [14].

II. PEXVIZ APPROACH

Our PexViz approach consists of three components: the Variant Control Flow Graph (VCFG) generator, the exploration-data augmentor, and the graph visualizer. The VCFG generator reads the program source code and transforms it into a VCFG representation, with an example shown in Figure 2. In a VCFG, a typical node corresponds to a branch statement in the program source code, and a directed edge between the starting node and the ending node indicates the control flow from the branch statement represented by the starting node to the branch statement represented by the ending node. The exploration-data augmentor is invoked by the IntelliTest exploration runtime and gathers useful information to augment the VCFG. Example information includes incremental path condition, being the predicate gathered from the branch statement corresponding to the current VCFG node, and flip count, being the count of flipping the constraint gathered from the incremental path condition of the current VCFG node. Finally, the graph visualizer reads the output VCFG and generates an interactive visualization front-end to present information to the users. We next illustrate the details of the graph visualizer, with a modified `BubbleSort` example.

A. Graph Visualizer

The graph visualizer includes the visualization front-end to display the VCFG graph and information on it, with an example shown in Figure 2. In particular, to improve the guidance provided by the visualization result, we present an interactive graph with rich information to help the users. We use different colors and shapes to encode the information that the VCFG nodes contain and to help the users easily differentiate the different situations represented by the VCFG nodes. In the graph, each VCFG node represents one branch statement from the source code. We extract and use the

Boolean predicate within the branch statement as the label for the VCFG node so that the users can quickly identify which line of code the branch statement belongs to. In case there are the same or similar branch conditions from the source code for multiple VCFG nodes, the users can click on each VCFG node to see the actual line number of the branch statement in the source code. Flip count is also shown on the VCFG node's label for convenience because it is an informative statistic in DSE exploration. The VCFG edges in the graph represent the execution flow of the program from one branch statement to another. Self edges and back edges are possible as well to indicate loops. The arrows on the VCFG edges indicate the direction of the execution flow. It is possible to have two-way VCFG edges between VCFG nodes. According to the data gathered in the exploration-data augmentor, the graph visualizer renders the information into filled colors, text labels, and textual data. In particular, the following information is visualized:

- Shape. A rectangle represents a VCFG node for a branch statement in the source code while a circle represents a utility VCFG node, such as an entry point.
- Filled color. (1) White represents that the incremental path condition in the VCFG node does not contain symbolic variables. White indicates lower inspection priority. (2) Green represents that the incremental path condition contains symbolic variables and has been reached at least once during the DSE exploration. Green is a safe color to indicate less threat to achieving code coverage. (3) Orange represents an un-flipped constraint from an incremental path condition that contains symbolic variables. Orange is a warning color to indicate a threatening factor. (4) Red represents an unreached branch statement during the DSE exploration. Red indicates a serious situation deserving attention. (5) Blue represents a utility VCFG node, such as an entry node, which is a node that does not come from the source code.

B. Example

Figure 1 shows a code snippet adapted from a bubble sort method in the open source DSA project (<https://archive.codeplex.com/?p=dsa>). We run IntelliTest with default settings on the code snippet and obtain 11/14 block coverage. The IntelliTest console result indicates that 122 paths have been explored until IntelliTest stops because of reaching the timeout boundary, and IntelliTest generates 6 inputs that cover different blocks, along with 2 inputs that trigger exceptions.

The tool users can investigate the corresponding PexViz graph as shown in Figure 2. There are 6 VCFG nodes in the PexViz graph, which has 97.8% fewer nodes than the 276 nodes from the SEViz [14] graph (not shown here due to space limit). The users can start examining the PexViz graph from the blue entry VCFG node. The users can directly observe the clear correspondence between VCFG nodes and branch statements through the VCFG node labels. Thus, such mechanism saves the users navigation time in contrast to clicking through each of the 276 nodes in the SEViz graph.

```

13  public void BubbleSort(int[] number, int n)
14  {
15      int count = 0;
16      for (int i = 0; i < number.Length;i++)
17      {
18          count++;
19          for (int j = 0;j < n;j++)
20          {
21              if (number[i] > n)
22              {
23                  int temp = number[j];
24                  number[j] = number[j + 1];
25                  number[j + 1] = temp;
26              }
27              if(count == 5)
28              {
29                  count = 0;
30              }
31          }
32      }
33      if (number.Length > 200)
34      {
35          throw new Exception("bug");
36      }
37  }
38 }
39 }
40 ...

```

Fig. 1: A code snippet of modified BubbleSort where a boundary problem is faced

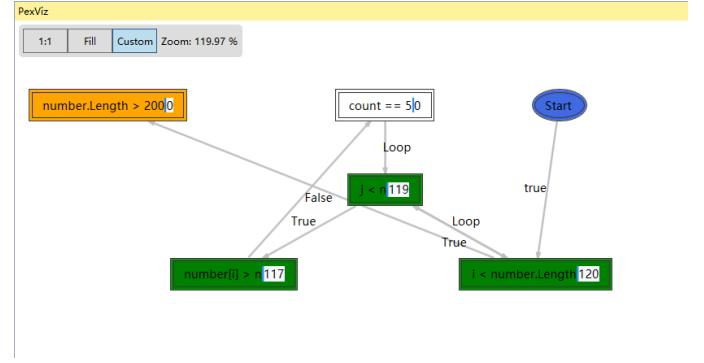


Fig. 2: PexViz visualization on running IntelliTest against the modified BubbleSort

The orange VCFG node showing 0 flip count immediately draws the users' attention with distinct color compared to all other VCFG nodes' colors. According to the orange VCFG node's label, the condition is expected to be evaluated to "True" if the length of the `number` array is larger than 200. To gain further understanding of the reason why the constraint is not flipped, the users can examine the three neighboring green VCFG nodes. All three green VCFG nodes have flip count of around 120. The information on the generated test inputs that the users can observe after clicking on the three green VCFG nodes indicates that the array with length larger than 200 is not created. IntelliTest stops before it is able to generate an array with length 200; therefore, increasing the bound of the maximum number of explored paths can be a solution to the problem. After the users increase the bound, IntelliTest manages to reach 14/14 (100%) block coverage.

Acknowledgment. This work was supported in part by National Science Foundation under grants no. CNS-1513939 and CNS-1564274.

REFERENCES

- [1] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, “EXE: automatically generating inputs of death,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, p. 10, 2008.
- [2] P. Godefroid, N. Klarlund, and K. Sen, “DART: Directed automated random testing,” in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*, 2005, pp. 213–223.
- [3] K. Sen, D. Marinov, and G. Agha, “CUTE: a concolic unit testing engine for C,” in *Proceedings of European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2005)*, 2005, pp. 263–272.
- [4] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, “Future of developer testing: Building quality in code,” in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER 2010)*, 2010, pp. 415–420.
- [5] N. Tillmann and J. De Halleux, “Pex – white box test generation for .NET,” in *Proceedings of International Conference on Tests and Proofs (TAP 2008)*, 2008, pp. 134–153.
- [6] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, “Fitness-guided path exploration in dynamic symbolic execution,” in *Proceedings of IEEE/IFIP International Conference on Dependable Systems & Networks (DSN 2009)*, 2009, pp. 359–368.
- [7] N. Tillmann, J. De Halleux, and T. Xie, “Transferring an automated test generation tool to practice: From Pex to Fakes and Code Digger,” in *Proceedings of ACM/IEEE international conference on Automated Software Engineering (ASE 2014)*, 2014, pp. 385–396.
- [8] X. Xiao, T. Xie, N. Tillmann, and J. De Halleux, “Covana: Precise identification of problems in Pex,” in *Proceedings of International Conference on Software Engineering (ICSE 2011)*, 2011, pp. 1004–1006.
- [9] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux, “Precise identification of problems for structural test generation,” in *Proceedings of International Conference on Software Engineering (ICSE 2011)*, 2011, pp. 611–620.
- [10] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su, “Synthesizing method sequences for high-coverage testing,” in *Proceedings of ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2011)*, 2011, pp. 189–206.
- [11] T. Xie, L. Zhang, X. Xiao, Y. Xiong, and D. Hao, “Cooperative software testing and analysis: Advances and challenges,” *J. Comput. Sci. Technol.*, vol. 29, no. 4, pp. 713–723, 2014.
- [12] X. Xiao, S. Li, T. Xie, and N. Tillmann, “Characteristic studies of loop problems for structural test generation via symbolic execution,” in *Proceedings of IEEE/ACM 28th International Conference on Automated Software Engineering (ASE 2013)*, 2013, pp. 246–256.
- [13] N. Li, T. Xie, N. Tillmann, J. d. Halleux, and W. Schulte, “Reggae: Automated test generation for programs using complex regular expressions,” in *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, 2009, pp. 515–519.
- [14] D. Honfi, A. Voros, and Z. Micskei, “SEViz: A tool for visualizing symbolic execution,” in *Proceedings of IEEE International Conference on Software Testing, Verification and Validation (ICST 2015)*, 2015, pp. 1–8.

Usability Challenges that Novice Programmers Experience when Using Scratch for the First Time

Yerika Jimenez
 Computer & Info. Science & Engineering
 University of Florida,
 Gainesville, USA.
 jimenyer@ufl.edu

Amanpreet Kapoor
 Computer & Info. Science & Engineering
 University of Florida,
 Gainesville, USA.
 kapooramanpreet@ufl.edu

Christina Gardner-McCune
 Computer & Info. Science & Engineering
 University of Florida,
 Gainesville, USA.
 gmccune@ufl.edu

Abstract—Block-based programming environments have increased students' interest in computer science (CS). Research suggests that block-based programming environments have positively impacted students' retention, effectiveness, efficiency, engagement, attitudes, and perceptions towards computing. We know that when novice programmers are learning to program in block-based programming environments, they need to understand the components of these environments, how to apply programming concepts, and how to create artifacts. However, few studies have been done to understand the impacts that usability of block-based programming environments may have on students' programming. In this poster, we present results from a two-part study designed to understand the impact that usability of the programming environment has on novice programmers when learning to program in Scratch. Our findings indicate that usability challenges may affect students' ability to navigate and create programs within block-based programming environments.

Keywords—Scratch, usability, block-based programming environments, computer science education

I. INTRODUCTION

Block-based programming environments have grown significantly since the 1980's from Turtle Graphics, a visual abstraction first implemented for the Logo language [1], to highly visual and interactive environments such as Scratch [2], Kodu [3], Pencil Code [4], and Blockly [5]. With the continued growth of block-based programming environments comes the challenge of understanding the usability challenges that novice programmers have when interacting with these environments. Kolling et al. suggest that it has become difficult for educational researchers to compare and judge the relative qualities of block-based programming environments and their respective suitability for a given context because of a lack of scientific validation or formal evaluation of these block-based programming environments [7].

Thus, conducting usability research on block-based programming environments is important for students, researchers, practitioners, and designers. The usability of these environments impacts first-time programmers because they have to learn the logic and syntax of computer science concepts and understand how to use the programming environment. As for researchers and practitioners, they need to understand the different usability problems within the environments and decide which of the environments will be the best to use for their students based on a given context and

age group. For the designers of block-based programming environments, it is important to understand the type of interactions that students experience while programming in these environments. The goal for the designers of these environments should be to ensure that usability issues in the environment do not negatively affect students' programming experiences by increasing the cognitive load or mental effort. Thus, we designed a two-part study which aims to understand the usability challenges that first-time programmers experience when using Scratch for the first time.

II. METHODOLOGY

The studies and results presented in this poster focus on answering the following research question: What are the kinds of usability challenges that first-time programmers initially encounter when creating artifacts in Scratch?

The studies were run at a large public research university in the United States in Spring 2017 and were part of a larger study that focused on understanding the usability of Scratch (Part 1) and assessing first-time programmers' challenges and mental effort when they code using Scratch (Part 2).

A. Participants

Participants were recruited by email and word of mouth. We explicitly recruited participants with no programming experience. We had a total of 13 participants (ages 18–24): 10 females and 3 males. We had 4 White, 2 Hispanic, 4 African American, and 3 Asian-Pacific Islander participants. These participants were from 12 different majors: microbiology (2), cell science, art history, mathematics, journalism, pre-nursing, pre-pharmacy, chemistry, biological engineering, civil engineering, environmental science, and nutritional sciences. All participants indicated in the pre-survey that they had no prior experience in computer science or programming.

B. Procedure

In the Usability study (Part 1), participants were instructed to perform six usability tasks to help researchers identify potential usability challenges within Scratch 2.0 (Table 1). These are common tasks that all Scratch users perform repeatedly when creating Scratch projects or beginning to program in Scratch. These tasks also consist of interactions

that participants needed to understand to complete their programming tasks in the Programming Study (Part 2).

TABLE I. USABILITY TASKS DESCRIPTION

Task	Description
1	Login to Scratch
2	Create a new project in Scratch
3	Upload a sprite and a background
4	Customize the sprite costume and delete the Scratch Cat
5	Change the size of the sprite
6	Make the sprite move

In Part 2, students learned four CS concepts (initiation, sequencing, iterations, and sensing) from watching brief videos and then creating a Dancing Ballerina Animation in Scratch. The program was broken down into five tasks designed to help the students create a ballerina performance. Each of the tasks built upon the previously completed task and allowed students to demonstrate their understanding of the concepts as they created the animation (Table 2).

TABLE II. PROGRAMMING TASKS

Tasks description	Computer science concept
Video 1: Scratch Overview & Video 2: Event Handling & Sequencing	
Task 1 - Setup stage	Initiation & event handling
Video 3: Iteration & Sensing	
Task 2 - Program dance sequence	Sequencing & iteration (loops)
Task 3 - Prevent the ballerina from running off stage	Conditionals (Sensing), sequencing
Task 4 - Add music	Initiation, event handling, sequencing, sensing, iteration,
Task 5 - Synchronize music, stage, and ballerina	parallelism

C. Data Collection & Data Analysis

To answer our research question, we collected and analyzed retrospective think aloud (RTA) and observations data. RTA videos were recorded using the software InqScribe and we used inductive coding and grounded theory [8] to categorize the students' responses.

III. RESULTS

Our results show that the users' (novice programmers) had usability challenges while using Scratch 2.0.

Usability Issue #1: Components/sections with small iconography or text labels led users to often overlook them in the interface which leads to frustration/inability to complete simple programming tasks.

Design Implication #1: Increasing the font size and bolding icons and words would allow users to locate icons and words faster. Moving the editing buttons to the middle of the interface will allow students to easily identify them.

Usability Issue #2: Lack of user feedback led to students spending a significant amount of time looking for basic function features e.g. saving prompt and delete button.

Design Implication #2: Adding a "saving" prompt indicating a user when his/her program is being saved to address the user's concern about visual feedback for saving.

Design Implication #3: Adding a delete button with a "trash" icon would allow participants to visually identify how to delete a sprite or background.

Usability Issue #3: Switching between different sprites to visualize program execution during debugging and synchronization obstructed users' ability to resolve program bugs. Users resorted to copying and pasting code from multiple sprites into one so that they could observe their program's behavior. But they did not realize that they were changing the behavior of the sprites with this additional code.

Design implication #4: Support for visualizing program execution block by block. By providing block by block execution, novice programmers will be able to understand how the program is being executed at a block level and better visualize the debugging process.

IV. CONCLUSION

Conducting comprehensive usability studies and publishing usability results is essential in informing and refining the design of block-based programming environments in order to improving the user experience. With millions of students using these environments, it is important to understand their usability and effectiveness to inform which environments researchers and practitioners choose given a context or an age group. This poster focused on identifying the usability challenges that students' experience when using Scratch for the first time. One of the major contributions of this poster is the identification of challenges students had while trying to understand and debug their programs. An example of this is that the novices were unable to see highlights of individual code blocks when executing their scripts or the visualization of parallel execution of scripts across multiple sprites.

REFERENCES

- [1] S. Papert, "Mindstorms: children, computers, and powerful ideas,". New York: Basic Books, 1980.
- [2] M. Resnick, J. Maloney, H. Andres, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, and Y. Kafai, "Scratch: Programming for All," in Communications of the ACM, 52(11), pp. 60–67, Jan. 2009.
- [3] M. MacLaurin, "Kodu: end-user programming and design for games", in Proceeding FDG '09 Proceedings of the 4th International Conference on Foundations of Digital Games, Orlando, Florida, 2009.
- [4] "Pencil Code", Pencilcode.net, 2017. [Online]. Available: <https://pencilcode.net/>. [Accessed: 23- Jul- 2017].
- [5] "Blockly Games", Blockly-games.appspot.com, 2017. [Online]. Available: <https://blockly-games.appspot.com/>. [Accessed: 23- Jul- 2017].
- [6] D. Weintrop and U. Wilensky, "To block or not to block, that is the question," in Proc.of the 14th International Conference on Interaction Design and Children, 2015, pp. 199–208.
- [7] M. Kölling and F. McKay, "Heuristic Evaluation for Novice Programming Systems", ACM Transactions on Computing Education, vol. 16, no. 3, pp. 1-30, 2016.
- [8] J. Corbin and A. Strauss, Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory, 3rd ed. Thousand Oaks,CA:Sage,2008.

BioWebEngine: A generation environment for bioinformatics research

Paolo Bottoni

*Dep. of Computer Science
Sapienza University of Rome
Rome, Italy
bottoni@di.uniroma1.it*

Tiziana Castrignanò

*SCAI department
CINECA
Rome, Italy
t.castrignano@cineca.it*

Tiziano Flati

*IBIOM
National Research Center (CNR)
Rome, Italy
t.flati@cineca.it*

Francesco Maggi

*Dep. of Computer Science
Sapienza University of Rome
Rome, Italy
maggi.1209338@studenti.uniroma1.it*

Abstract—With technologies for massively parallel genome sequencing available, bioinformatics has entered the “big data” era. Developing applications in this field involves collaboration of domain experts with IT specialists to specify programs able to query several sources, obtain data in several formats, search them for significant patterns and present the obtained results according to several types of visualisation. Based on the experience gained in developing several Web portals for accessing and querying genomics and proteomics databases, we have derived a meta-model of such portals and implemented BioWebEngine, a generation environment where a user is assisted in specifying and deploying the intended portal according to the meta-model.

Index Terms—Bioinformatics, MDD, visualisation.

I. INTRODUCTION

With the advent of new technologies on massively parallel sequencing, the size and number of experimental raw sequencing datasets available has increased exponentially, thus ushering bioinformatics into the “big data” era. Web portals associated with “omics” databases have therefore become the primary entry points for accessing and querying such huge amount of biological data and for retrieving biological information. In this context, the development of applications supporting researchers in bioinformatics involves tight collaboration with IT specialists to present the obtained results according to several types of visualisation.

Based on the experience gained in developing such environments [1], [2] we have implemented BioWebEngine (BWE), a generation environment where a user is assisted in specifying the configuration of a Web portal via a set of templates derived from a meta-model, which abstracts from and generalises features of previously developed portals.

The main features of the proposed framework are:

- 1) Versatility: BWE can produce various types of portal.
- 2) Ease of use: BWE is aimed at non-programmers.
- 3) Separation of concerns: designed as Single Page Applications, front-end and server-side logics are separated and communicate by means of a customizable API-based infrastructure.
- 4) Component dynamicity: the Web portals created with BWE are dynamic in each and across sub-components.

II. GENERATING PORTALS FROM TEMPLATES

The construction of the meta-model of Web portals for omics databases started from the analysis of existing portals [3], [4] and of the code for configuring them. As this code had been developed manually and independently, we looked for potential discrepancies in the descriptions of elements of the same type, as well as for critical issues or weaknesses. This analysis resulted in the definition of a meta-model (to which portals should conform) which provides a blueprint for the interactive generation of configuration files, keeping them as close as possible to the existing format, while overcoming problems and limitations identified during the analysis.

Starting from the meta-model thus obtained, we derived a collection of templates, one for each specific component in the textual (JSON) configuration file, corresponding to the types defined in the meta-model, based on a custom conversion mechanism to achieve the best fit to our implementation requirements. The whole derivation process was designed in accordance with the following fundamental principles. All meta-model concrete classes have been transformed in objects in the template and class attributes converted into objects’ fields. Abstract classes do not undergo a formal conversion and are thus discarded. If a given attribute is associated with a certain enumeration, the admissible values of the corresponding object field are those provided by the enumeration. Operation members are implemented by a corresponding API. Associations between classes (simple associations, aggregations and compositions) are resolved as fields of the containing objects.

BWE consists of two main subsystems: the Web portal generator (WPG) and the Visualization engine (VE), both interfacing through a server providing access to a JSON configuration file. Users exploit WPG to define a portal by populating the configuration file. In WPG, a *user layer* contains a series of handlers by which users can define the entities in the portal; an *application layer* contains the modules that inject the user information into the template components and subsequently save the populated templates. The overall interaction with WPG is based on a form-filling paradigm, which proved to be intuitive to users, relieving users from learning a domain specific visual language. The VE subsystem is responsible for rendering of the portal and management

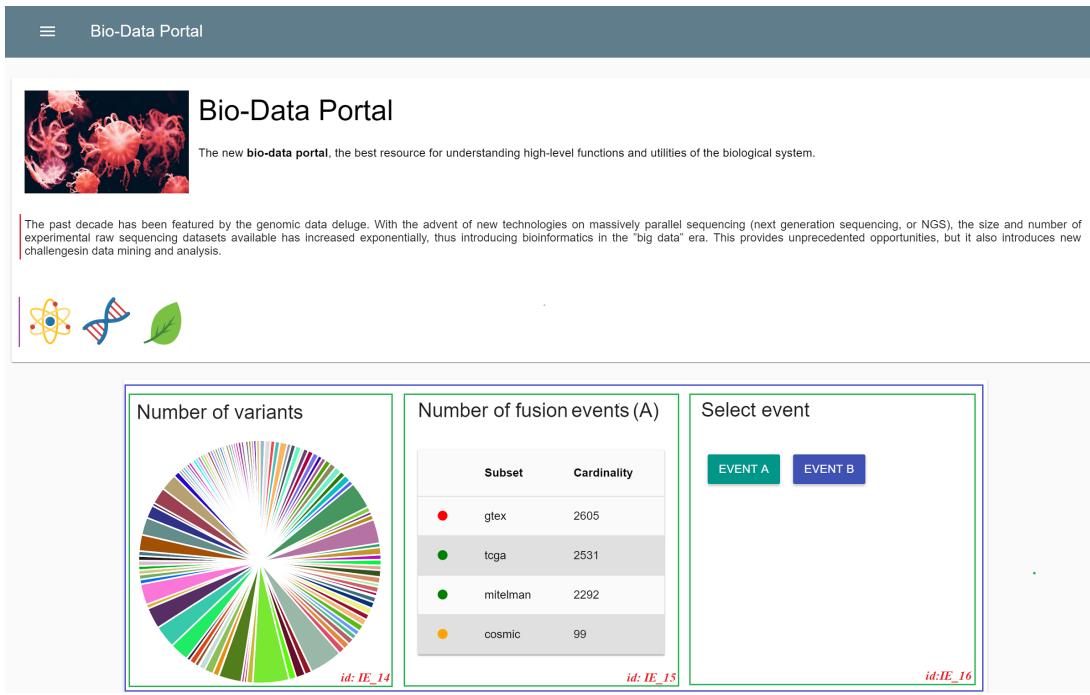


Fig. 1: Excerpt of the execution under the Visualization engine subsystem of the portal.

of its dynamics. This includes: i) parsing and rendering the portal information retrieved from the server; ii) taking charge of the portal interactivity, including both user navigation and the complex handling of *TriggerTypes* and *Actions*; iii) coordinating the communication with data centers in order to update or populate the portal's elements.

Notably, since WPG and VE communicate with endpoints through AJAX calls, also the information returned by data repositories is supposed to be returned in a JSON format compliant with that expected by VE in order to render such information and update the involved components. This is of extreme importance, since not only does this assure a separation of concerns, but it also fosters modularity.

III. TESTING THE GENERATION MECHANISM

In order to demonstrate the capabilities of BWE, we have recreated the PeachVar-DB and LiGeA portals, specifying them in WPG and enacting them with VE.

Figure 1 shows an example of the portal created through the interaction with WPG. As can be seen, under the header there are several elements: a pie-chart, a table and two buttons, all laid out in a grid-layout of three columns and one row.

As mentioned before, the definition of models for PeachVar-DB and LiGeA portals had been provided manually. It took about 4 months for PeachVar-DB portal¹ and about 8 month for the LiGeA portal² before they could be considered full-fledged products. The development of BWE itself, instead, was

¹Released in December 2017, PeachVar-DB has received about 180 unique visits so far.

²Released in January 2018, LiGeA has received about 30 unique visits so far.

carried out in less than 4 months, and then reproducing the PeachVar-DB portal with BWE took only a few days.

IV. CONCLUSIONS

We have presented BioWebEngine, a generation environment assisting users in specifying and deploying Web portals oriented to interaction with 'omics databases based on a collection of templates derived from a meta-model of such portals. BWE can be characterised as: versatile, easy to use, split in logic layers, and dynamic in each sub-system. All these features combined together allow users to configure and implement their Web portals with considerable less effort than required in standard development procedures. Future work might include handling arbitrarily complex actions, graphic structures or dynamic objects.

ACKNOWLEDGMENT

The authors would like to thank the Italian Node of Elixir project (<http://elixir-italy.org>) for the availability of high performance computing resources and support.

REFERENCES

- [1] T. Castrignanò *et al.*, “The PMDB protein model database,” *Nucleic Acids Research*, vol. 34, no. suppl_1, pp. D306–D309, 2006. [Online]. Available: <http://dx.doi.org/10.1093/nar/gkj105>
- [2] T. Castrignanò *et al.*, “ASPicDB: A database resource for alternative splicing analysis,” *Bioinformatics*, vol. 24, no. 10, pp. 1300–1304, 2008. [Online]. Available: <http://dx.doi.org/10.1093/bioinformatics/btn113>
- [3] M. Cirilli *et al.*, “PeachVar-DB: A curated collection of genetic variations for the interactive analysis of peach genome data,” *Plant and Cell Physiology*, vol. 59, no. 1, p. e2, 2018. [Online]. Available: <http://dx.doi.org/10.1093/pcp/pcx183>
- [4] S. Gioiosa *et al.*, “Massive NGS data analysis reveals hundreds of potential novel gene fusions in human cell lines,” *GigaScience*, 2018.