

©Copyright 2020

Jane Hoffswell

Languages and Visualization Tools
for Data-Centric End-User Programming
of Interactive Visualization Designs

Jane Hoffswell

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2020

Reading Committee:

Jeffrey Heer, Chair

Alan Borning

Amy J. Ko

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Languages and Visualization Tools
for Data-Centric End-User Programming
of Interactive Visualization Designs

Jane Hoffswell

Chair of the Supervisory Committee:
Professor Jeffrey Heer
Paul G. Allen School of Computer Science & Engineering

Visualizations can facilitate data exploration and communication of insights. While many tools exist to support the design of interactive visualizations, the development process relies heavily on the user’s domain and programming expertise. To facilitate interactive visualization design, improved tools should better align with and enrich the user’s mental model. This dissertation contributes three projects to help end-user programmers more effectively *author*, *understand*, and *reuse* both code and data to design interactive visualizations. To this end, this dissertation explores (1) the design of customized graph layouts, (2) the development and debugging process for interactive visualizations, and (3) the synchronization and customization of multiple visualization versions for responsive visualization design.

Across these projects, this dissertation explores how new techniques that raise the level of abstraction can help users focus on the domain-specific concepts of interest, while deferring low-level implementation details to the underlying system. A crucial step in this process is to identify and communicate actionable information to the end user. To accomplish this goal, this dissertation contributes three sets of formative interviews with potential users to identify unique challenges and design opportunities for the given domain. These interviews illustrate the disconnect between users’ expectations and the functionality provided by existing systems or development workflows, and further highlight the types of tangential, low-level information that systems should hide from the user’s view to improve the development process. While this information may be useful for completely or accurately representing the program behavior, such details can unnecessarily complicate the program understanding or debugging process.

Motivated by these challenges and interviews, this dissertation contributes new programming languages and program visualization tools to better help end-user programmers understand the underlying system behavior. These approaches raise the level of abstraction to reflect the user’s unique domain expertise and obfuscate unnecessary system details. To this end, the proposed techniques aim to communicate relevant and actionable information to the user, and better prioritize the user’s most important development tasks.

This dissertation first contributes SetCoLa: a domain-specific language for custom graph layout that leverages high-level constraints to encode the user’s domain knowledge. SetCoLa facilitates code authoring and reuse by reducing the number of user-authored constraints by one to two orders of magnitude. However, the declarative nature of this language requires users to map between their high-level input and the system-produced output to debug or interpret the behavior. To explore the unique challenges and novel solutions for program understanding of declarative languages, this dissertation next turns to Vega: a declarative grammar for interactive visualization design. This dissertation then contributes a series of three projects for program understanding in Vega, which evolve to address the unique development needs of users at different stages in the development process. These techniques support (1) low-level system development via a data flow graph visualization, (2) debugging interactions with visualizations of contextually relevant details, and (3) unobtrusively revealing details of the runtime behavior during both normal execution and debugging. Whereas Vega focuses on the design of a single interactive visualization, responsive visualizations require designers to develop multiple concurrent designs that adapt based on the screen size or interactive capabilities of the end user’s device. To support this process, this dissertation contributes four design guidelines and a set of core system features for a responsive visualization design system that supports simultaneous editing and device-specific customizations.

For each of these projects, this dissertation further contributes evaluations of this work via user studies or reproductions of real-world examples. The user evaluations demonstrate the utility of the proposed approaches for improving how end users interact with and understand the system functionality, whereas the reproductions illustrate the flexibility and expressiveness of the proposed techniques. Overall, this dissertation aims to better understand people and to help people better understand systems. This dissertation contributes novel techniques to support end-user programmers in developing, understanding, and debugging custom interactive visualization designs, and suggests new avenues for future work.

TABLE OF CONTENTS

	Page
List of Figures	v
Chapter 1: Introduction	1
Thesis Statement	2
Challenge 1: Raise the level of abstraction to reflect user expertise.	3
Challenge 2: Communicate system behavior as actionable information.	3
Challenge 3: Support the tasks that matter most to the user.	3
1.1 Thesis Contributions	4
1.2 Thesis Outline	9
1.3 Prior Publications and Authorship	11
Chapter 2: Background and Related Work	12
2.1 Visualization Design Systems	13
2.2 Graph Visualization Techniques	15
2.3 Visualizations to Facilitate Program Understanding and Debugging	16
2.4 Text and Environment Augmentation with Visualizations	18
2.5 Empirical Studies of Programmers and Program Understanding	18
2.6 Declarative Programming Languages and Debugging	19
2.7 Domain-Specific Programming Languages	20
2.8 Discussion and Applicability of Related Work	21
Chapter 3: Understanding the Program Behavior of Constraint Systems	22
3.1 Related Work: Constraint Programming Systems	23
3.2 Formative Interviews: Utilizing and Understanding Constraints	25
3.3 Limitations and Future Work	30
3.4 Summary of Contributions	31

Chapter 4:	Authoring and Reusing Domain-Specific Graph Layouts with SetCoLa	32
4.1	Related Work: Domain-Specific Graph Visualization	35
4.2	Design of SetCoLa: A Set-Based Constraint Layout for Graphs	36
4.3	Evaluation: Real-World Examples Reproduced in SetCoLa	50
4.4	Limitations and Future Work	60
4.5	Summary of Contributions	64
Chapter 5:	Program Understanding in Vega: A Declarative Visualization Grammar	65
5.1	Related Work: Functional Reactive Programming	66
5.2	Background and Terminology for the Vega Visualization Grammar	67
5.3	Visualizing the Vega Runtime Behavior as a Data Flow Graph	69
5.4	Formative Interviews: Understanding Declarative Visualization Design	73
5.5	Summary of Contributions	76
Chapter 6:	Visual Debugging Techniques for Reactive Data Visualization	77
6.1	Design of Visual Debugging Techniques for Program Understanding	78
6.2	Evaluation: Debugging Faulty Visualizations	85
6.3	Limitations and Future Work	92
6.4	Summary of Contributions	94
Chapter 7:	Augmenting Code with In Situ Visualizations	95
7.1	Design Space of Code-Embedded Visualizations	97
7.2	Implementation of Code Augmentations for the Online Vega Editor	111
7.3	Evaluation: Understanding Program Behavior of Vega	113
7.4	Limitations and Future Work	120
7.5	Summary of Contributions	122
Chapter 8:	Authoring and Reusing Responsive Visualization Designs	123
8.1	Related Work: Responsive Web Design and Mobile Visualization	126
8.2	Formative Interviews: Responsive Visualization Design Practices	127
8.3	Techniques for Flexible Responsive Visualization Design	132
8.4	Evaluation: Reproducing Real-World Responsive Visualizations	143
8.5	Limitations and Future Work	149
8.6	Summary of Contributions	151

Chapter 9: Conclusion	152
9.1 Summary of Contributions	153
9.2 Discussion and Reflections on Three Core Dissertation Challenges	154
Challenge 1: Raise the level of abstraction to reflect user expertise.	154
Challenge 2: Communicate system behavior as actionable information.	156
Challenge 3: Support the tasks that matter most to the user.	157
9.3 Future Research Directions	160
9.4 Concluding Remarks	162
 Bibliography	 163
 Appendix A: Interview Resources: Understanding the Behavior of Constraint Systems	185
A.1 Formative Interview Screening Survey	185
A.2 Formative Interview Script Template	186
 Appendix B: Historical Debugging Approach for Vega using the JavaScript Console	189
 Appendix C: Interview Resources: Visualizing Vega’s Behavior as a Data Flow Graph	197
C.1 Formative Interview Script Template	197
C.2 Data Flow Example Visualizations	199
 Appendix D: Evaluation Resources: Visual Debugging Techniques for Vega	203
D.1 Evaluation Post-Task Survey and Exit Survey	203
D.2 Evaluation Reference Sheet	205
 Appendix E: Evaluation Resources: Augmenting Code with In Situ Visualizations	206
E.1 Evaluation Screening Survey	206
E.2 Evaluation Script	208
E.3 Evaluation Instruction Sheet	211
E.4 Evaluation Instruction Sheet for the In Situ Visualizations	215
E.5 Evaluation Training Tasks	215
E.6 Task-Specific Program Understanding Questions	216
E.7 Evaluation Exit Survey	219

Appendix F: Interview Resources: Responsive Visualization Design Practices	221
F.1 Formative Interview Script Template	221
Appendix G: Responsive Visualization Corpus Bibliography	224
Appendix H: Responsive Visualization Examples	229
H.1 “Total Cost of Major Natural Disasters”	229
H.2 “Incidents at Sea”	231
H.3 “In close decisions, Kennedy voted in the majority...”	232
H.4 “Percentage of the population without access to improved water”	233
H.5 “Activity at the time of spill”	235
H.6 “Was Yahoo Late to Mobile?”	236
H.7 “Beijing Air Quality Index (PM2.5)”	237

LIST OF FIGURES

Figure Number	Page
4.1 The TLR4 biological system layout using Cerebral [6] and SetCoLa [83]. . . .	33
4.2 The SetCoLa spec and resulting WebCoLa constraints for a small tree layout.	37
4.3 The number of nodes, links, and constraints for each SetCoLa example. . . .	44
4.4 The syphilis network layout from Rothenberg et al. [164] and SetCoLa. . . .	51
4.5 The SetCoLa specification for the syphilis social network in Figure 4.4. . . .	52
4.6 The SetCoLa specification for the TLR4 biological system in Figure 4.1. . . .	53
4.7 A generic SetCola specification for biological networks from InnateDB. . . .	54
4.8 The layout for the TLR4 biological system using InnateDB and SetCoLa. . .	55
4.9 The layout for the DDX58 biological system using InnateDB and SetCoLa. .	55
4.10 The layout for the NOD-like signaling pathway using InnateDB and SetCoLa.	55
4.11 The layout for the MAPK1 biological system using InnateDB and SetCoLa. .	56
4.12 A simple Kruger National Park food web layout [124] reproduced in SetCoLa.	57
4.13 The SetCoLa specification for the Kruger National Park layout in Figure 4.12.	57
4.14 The Serengeti food web layout from Baskerville et al. [7] and SetCoLa. . . .	58
4.15 The SetCoLa specification for the Serengeti food web in Figure 4.14.	59
5.1 The Vega specification and simplified data flow graph for a grouped bar chart.	70
5.2 The prototype data flow graph for a grouped bar chart.	71
6.1 The visual debugging techniques and development environment for Vega. . .	80
6.2 The behavior of the overview, timeline, and signal annotations for debugging.	81
6.3 The user study methodology used to evaluate our visual debugging techniques.	85
6.4 Relevant program states and code for debugging an interactive index chart. .	87
6.5 Relevant program states and code for debugging panning on a scatterplot. .	88
6.6 Relevant program states and code for debugging brushing on a scatterplot. .	90
6.7 Average ratings for each visual debugging technique from the user evaluation.	91
7.1 The design space of code embedded visualizations.	96

7.2	An example set of code augmentations demonstrating object simplification. . .	99
7.3	An example set of code augmentations demonstrating linked filtering.	106
7.4	Twelve placement techniques for code augmentations.	108
7.5	Example code augmentations for an index chart in the online Vega editor. . .	112
7.6	The user study methodology used to evaluate code augmentations for Vega. .	115
7.7	The helpfulness, interpretability, and intrusiveness of the code augmentations.	118
8.1	An example visualization annotated with the responsive techniques used. . .	124
8.2	The sources and visualization types in our responsive visualization corpus. .	132
8.3	The responsive visualization techniques used for mobile devices.	133
8.4	The overview of our responsive visualization system.	136
8.5	System panels showing the design variation for a responsive visualization. . .	141
8.6	System panels for interaction design in our responsive visualization system. .	142
8.7	Steps to recreate a responsive visualization from Reuters Graphics [G52]. . .	145
B.1	The historical Vega development and debugging environment.	189
B.2	The broken and correct version of an index chart visualization.	190
B.3	Screenshots of how to debug Vega using the JavaScript console.	191
B.4	Screenshots from the Vega specification for an index chart.	192
B.5	More screenshots of how to debug Vega using the JavaScript console.	193
B.6	More screenshots from the Vega specification for an index chart.	194
B.7	The Vega code to fix the broken index chart.	196
C.1	The data flow graph for a grouped bar chart.	199
C.2	The specification, output, and annotated data flow graph.	200
C.3	The specification, output, and simplified data flow graph.	201
C.4	The annotated specification, output, and simplified data flow graph.	202
D.1	A reference sheet of the visual debugging techniques available for Vega. . . .	205
E.1	The Vega example for the training task in the code augmentation evaluation.	212
E.2	Tooltips added to the online Vega editor to evaluate the code augmentations.	214
E.3	Descriptions of the in situ visualizations used in the user evaluation.	215
H.1	Reproductions of a New York Times visualization [G13].	230
H.2	Reproductions of a Reuters Graphics visualization [G52].	231
H.3	Reproductions of a New York Times visualization [G36].	233

H.4	Reproductions of a National Geographic visualization [G50].	234
H.5	Reproductions of a Reuters Graphics visualization [G52].	235
H.6	Reproductions of a Harvard Business Review visualization [G19].	236
H.7	Reproductions of a ChartAccent [159] and Vega-Lite [176] visualization. . . .	237
H.8	Responsive visualization codes for our reproduced examples.	238

ACKNOWLEDGMENTS

This dissertation is the culmination of many years of work and collaboration. I am both excited and humbled to have reached this milestone, and so incredibly thankful for all of the support I have received along the way. To all of the wonderful people that have supported, encouraged, and cheered me on, thank you.

I would like to take a moment to give some special thanks to my advisor, Jeffrey Heer. Jeff has supported me throughout the entirety of my grad school career and I am absolutely honored to have been a part of his lab at the University of Washington. Over the years, Jeff has always encouraged me to explore new and exciting topics, and has helped guide me through my evolving research interests to ultimately grow into a confident and independent researcher. I am certain that Jeff has had and will continue to have a lasting impact on my development as a researcher, and for that I am incredibly thankful. Beyond research, Jeff has created a lovely community of students and colleagues. I have thoroughly enjoyed our many group potlucks and happy hours, as great opportunities to relax and celebrate the achievements of our group.

I would also like to thank my many incredible research collaborators and mentors. During my time at the University of Washington, I have had many wonderful opportunities to learn from and be inspired by the amazing people from within our school, and from the larger computer science community. The work in this dissertation is a testament to many of these impactful collaborations. I would first like to thank Alan Borning for his support and collaboration on SetCoLa and for fostering my interest in constraints, as well as for his continued enthusiasm as part of my dissertation

committee. I would like to thank Arvind Satyanarayan for his mentorship and collaboration on my program understanding work for Vega; during my first few years of grad school, Arvind encouraged and challenged me as a young researcher to step outside my comfort zone and to establish my role in the computer science research community. I would like to thank my mentors from Adobe Research, Zhicheng Liu and Wilmot Li, for their collaboration on my responsive visualization work. My time as an intern at Adobe was a uniquely valuable experience that enabled me to explore a whole new set of research topics and to explore opportunities for research impact in industry. I would also like to thank Amy Ko and Jevin West for participating on my dissertation committee. To my entire committee, thank you for your insights, critiques, and enthusiasm for this work that have helped push it to be stronger.

While my research collaborations played a crucial role in shaping my personal trajectory, they are just one piece of the overall experience. I would therefore like to thank my many friends and colleagues in the Interactive Data Lab and the UW Allen School who have spent time giving me feedback on papers, talks, and projects, and for providing me with much needed breaks from work and all around fun times. The tremendous amount of support from this community will have a lasting impact. My time at the University of Washington has been truly special and one I will certainly never forget.

Finally, I would like to acknowledge and give special thanks to my friends and family. I am so thankful for everyone that has come along with me on this remarkable journey. You have all supported and encouraged me along this path, through all of the good times and the challenges, and the experience has been that much better because of your support. I appreciate every ounce of confidence you have placed in me and I am excited to continue this journey with all of you as I move on to my major next steps.

Thank you.

Chapter 1

INTRODUCTION

Visualizations fulfill many different roles in data analysis and communication. Visualizations can provide insight during exploratory data analysis to confirm expectations about how the data should look, or to surface surprising or incorrect results. Visualizations can further communicate ideas for presentation or informational contexts, and may employ annotations to tell a particular story by directing the viewer’s attention to elements of interest. In both data analysis and communication, interaction plays a crucial role in visualization design as a way to support users in personally exploring the underlying data.

Depending on the purpose of the visualization, there are a variety of approaches for constructing the visualization design. For exploratory data analysis, tools like Tableau (formally Polaris [184]) and Voyager [206] support rapid visualization design via drag-and-drop in a graphical user interface, which is an approach known as the shelf construction model [175]. This method is great for exploring data by allowing users to rapidly change what information is visualized. Template-based approaches (e.g., [64, 131, 135, 142]) or domain-specific systems (e.g., [6, 16, 120, 181]) leverage a standard data format to easily or quickly produce visualization designs of a particular type, but do not otherwise support exploration of new visualization designs. While emphasizing ease of use, many of the aforementioned techniques offer only minimal support for customizing the visualization or interactive experience. To such ends, visual builders like Lyra [174], Data Illustrator [129], and Charticator [161] aim to support flexible visualization design via direct manipulation. These systems allow users to construct highly customized designs beyond what is possible with template based approaches or the shelf construction model. To support maximal flexibility of the design, approaches like D3 [23] and Vega [177] enable the construction of more complex and unique visualization de-

signs. Furthermore, both D3 and Vega enable the specification of end-user interactions with the resulting visualization. However, these approaches also require extensive programming expertise to effectively employ. In particular, the time-varying behavior of the visualization design can add additional complexity that the programmer must understand in order to effectively modify or debug the functionality of the resulting visualization.

Across these approaches, users are limited by what is possible in the system and may therefore produce visualization designs that reflect the system defaults rather than their personal preferences or goals. Users must try to adapt to the capabilities at hand in order to produce a visualization design that is at least close to their desired result. To further complicate this process, users often have their own unique, domain-specific expertise that informs their design goals and approach to programming problems [113]. This expertise is often underutilized or unsupported by the tools that users are trying to employ.

My research combines techniques from human-computer interaction (HCI), visualization, and programming languages to first identify the types of challenges that users face, then to develop and evaluate novel application solutions. By focusing new programming languages and end-user systems on the domain expertise and tasks most relevant to the user, we can improve how individuals interact with systems to better promote program understanding, and to proactively surface surprising or incorrect results. In particular, this dissertation explores the development process for how end users produce interactive visualization designs. Across the work described in this dissertation, I demonstrate the idea that:

THESIS STATEMENT

The design of new languages and program visualization tools that raise the level of abstraction from low-level system details to domain-specific concepts and operations for interactive visualization design can help end-user programmers more effectively *author*, *understand*, and *reuse* both code and data.

There are three core challenges that are central to the work explored in this dissertation that arise from the thesis statement. These challenges reflect common considerations explored in human-computer interaction research, but present particular challenges for how best to

identify and/or realize the appropriate approach for different contexts. I briefly introduce these challenges in the following paragraphs and further discuss the unique implications of these challenges to the projects in the individual chapters throughout this dissertation.

Challenge 1: Raise the level of abstraction to reflect user expertise.

The main approach proposed in the thesis statement and explored across this dissertation is to raise the level of abstraction from low-level system details to domain-specific concepts and operations. In other words, systems should take advantage of the unique knowledge and goals of the user to better support the tasks that matter most and the program understanding techniques that the user employs. The main challenge is therefore to identify what level of abstraction is appropriate for a given context and how best to encode user expertise.

Challenge 2: Communicate system behavior as actionable information.

To help end-user programmers with the task at hand, it is essential for systems to communicate the behavior as actionable information for the user. Rather than attempting to accurately reflect or provide insight into the full internal behavior, the system should prioritize surfacing relevant information that the user can immediately translate to the current development task. In other words, the system should avoid presenting too much information that will complicate how the user understands the behavior as related to their primary goals. A major challenge is therefore identifying what information *not* to show to the user.

Challenge 3: Support the tasks that matter most to the user.

Users employ their personal expertise to identify the high-level task on which to focus. To accomplish this task often requires navigating a series of sub-tasks, such as authoring, understanding, or reusing code and data. To provide a smooth development or design experience, systems should adapt to support the user's primary tasks while reducing the burden imposed by less essential tasks. However, it can be difficult to identify what task is most important to the user and how best to support users in accomplishing their goals. It can also be particularly challenging to balance techniques given that the user's goals may change.

In this dissertation, I will show that improved tools can better align with and enrich end-user programmers' mental models. In particular, this dissertation shows how new programming languages and tools that emphasize relevant domain-specific concepts—what data should be visualized or what tasks are most important—can help end-user programmers focus on the behaviors that matter most to them while leveraging their personal expertise. For programming contexts in particular, I show that by visualizing program state in situ, end-user programmers will be more aware of the impact of code changes and thus reduce the time spent ignorant of errors or switching between tasks [87, 88]. When developing new programming languages or end-user facing systems, I show that raising the level of abstraction can help domain experts encode their expertise while transferring responsibility for unessential implementation details to the underlying system [83, 84].

1.1 Thesis Contributions

Visualizations can help people effectively analyze and communicate information. To support the design of data-rich visualizations, numerous visualization tools and languages have been developed, for example [23, 129, 161, 174, 176, 177, 184]. However, the process of designing highly customized, domain-specific visualizations often requires programming and/or domain expertise. This dissertation contributes three projects that help end-user programmers more effectively *author*, *understand*, and *reuse* both code and data for interactive visualization design. Across these projects, I explore how techniques that raise the level of abstraction can help end users focus on the domain-specific concepts of interest, while deferring low-level implementation details to the system. I also consider the design of new visualization strategies to support users in understanding the system functionality and code behavior.

This dissertation first explores the design of customized graph visualizations. An effective graph layout can reveal complex properties of the underlying structure, such as the hierarchy or network connectedness. Graph visualizations can further provide unique, domain-specific insights when the underlying layout appropriately reflects the important domain-specific details of the data. For example, in a biological pathway, nodes can be layered by their sub-

cellular location to visualize the network relationships in the context of the cellular structure. For this use case, the visualization tool Cerebral [6] was specifically designed to handle such networks. However, when a customized tool does not already exist for the domain of interest, domain experts must fall back on ill-fitting techniques or develop a customized algorithm of their own. One way in which to encode some flexibility in the graph layout is using constraints; constraints combine the ease of an automatic layout approach with the customizability needed for domain-specific layouts. However, low-level constraint approaches often require developers to write and maintain constraints between individual nodes, thus resulting in hundreds of constraints even for small graphs. Furthermore, these constraints only apply to the individual nodes for which they were specified. To create a similar layout for a different graph requires specifying a new set of constraints for the new graph.

To better understand the many complexities and advantages of constraints, this dissertation first contributes a preliminary set of interviews with programmers developing constraint-based applications. These interviews highlight the fact that constraints can be perfect for applications in which they are appropriate, but can introduce unnecessary overhead in tasks where a more straightforward algorithm or approach would be ideal. Furthermore, across all participants, the process of learning and debugging the behavior of constraints proved to be a substantial challenge and vastly undersupported by existing tools. Despite these challenges, constraints can prove to be a natural way in which to express requirements and flexibility in a system, and have been used for a variety of contexts such as scheduling [54], user interface design [2, 38, 187, 188], and visualization layout for graphs [41, 43, 48] and charts [161].

To address the challenges in low-level constraint systems for graph layout, I contribute SetCoLa: a new language for specifying high-level constraints for customized, domain-specific graph layout. SetCoLa supports domain experts in *authoring* customized layouts by allowing them to focus on how the layout corresponds to particular domain-specific properties of the graph. Whereas prior approaches utilized node-level constraints between individual pairs of nodes, SetCoLa reduces the number of user-authored constraints by one to two orders of magnitude. Furthermore, by abstracting the application of constraints to apply to domain-

specific properties, SetCoLa allows the customized layout to be *reused* across multiple graphs in the same domain. Finally, the constraints themselves are more *understandable* because they reflect the relationships within the data, rather than between individual pairs of nodes. I demonstrate the conciseness, generalizability, and expressiveness of SetCoLa on a series of real-world examples from ecological networks, biological systems, and social networks.

To consider a wider class of visualizations, I next turn to Vega [177]: a declarative visualization grammar for interactive visualization design. Vega emphasizes domain-specific visualization constructs to allow end-user programmers to focus on how a visualization should be designed, while deferring the low-level implementation details to the underlying system. In other words, the user may focus on high-level constructs—what data should be visualized and in what way—without needing to control exactly how individual data elements are connected to visual elements. While this abstraction allows end-user programmers to employ their expertise and focus on the constructs of interest, the disconnect between the code and output can prove challenging when it comes to understanding faulty visualization behaviors. For this project, I first explore the design of a data flow graph visualization of the underlying system behavior. In a set of interviews with expert Vega users, participants noted that while this visualization can be useful for Vega system developers, it provides too much internal information tangential to their end-user debugging tasks. Instead, user facing tools should focus on the smaller fraction of *actionable* functionality relevant to the task at hand.

To address the needs of Vega end users, this dissertation next contributes a set of visual debugging techniques for reactive data visualization that aim to improve program *understanding*. These techniques support the refinement of the end-user programmer’s mental model through exploration of both the data and program state. To enable inspection of the state and the behavior of changes over time, I introduce three elements: a timeline of interactive signals which shows the time-varying behavior of the Vega output; annotations of relevant encodings in situ on the output to show how data maps to visual elements; and a dynamic data table showing the distribution and variation of data over time. These techniques provide interpretable insights into core Vega elements that were otherwise hard to

grasp or recall. In an evaluation with first-time Vega users, I show that participants could effectively understand the source code to accurately identify bugs or crucial dependencies.

While these techniques can prove essential to tracking problematic behaviors in the code, users sometimes had a hard time identifying where or when to look for a particular piece of information. Similar to other programming environments, the visual debugging techniques were split across multiple, coordinated views. However, studies have shown that switching between system views imposes a burden on programmers, making it difficult for them to maintain a clear picture of the overall context of the runtime behavior [118, 147, 151, 167]. While my proposed techniques can help reduce the gap between the Vega code and resulting output, opportunities still remained to better connect the debugging tools and code.

To help reduce the separation between code and the program understanding techniques, this dissertation further contributes a design space of program visualizations that can be embedded directly within the source code. These in situ visualizations similarly aim to support program *understanding* while also allowing end-user programmers to focus on code *authoring*. Instead of requiring end-user programmers to shift their focus to a separate view, the in situ visualizations act as a preemptive way to surface important information from the otherwise hidden program state and to display it within the context of the code itself. In a follow-up evaluation, I show that the embedded visualizations can help first-time Vega programmers improve their score on a set of program understanding questions. These techniques also exhibit positive effects on participants' self-reported speed and accuracy.

The proposed visual debugging techniques and in situ visualizations each surface hidden details of the Vega system behavior to facilitate the design of interactive visualizations. For some contexts however, the design of a single visualization is just one piece of the process. When used in news articles, for example, journalists often need to develop multiple versions of a visualization so as to support a responsive design that adapts to the device context—such as the screen size or the interactive capabilities of the device. To better understand the design process for responsive visualizations, this dissertation contributes semi-structured interviews with journalists and a survey of responsive visualization techniques commonly used

by news outlets. Based on these results, I found that despite readers increasingly consuming news content on mobile devices, many journalists still follow a desktop-first approach; the rationale for this discrepancy reflects the perceived opportunities and defaults of existing visualization construction systems rather than the values of the journalists themselves.

To better support responsive visualization design, I further contribute four design guidelines and a set of core system features that allow designers to view, create, and modify multiple device-dependent visualizations simultaneously. To ensure that users can easily understand the differences between their designs, the system foregrounds variation to support propagation of changes between visualization designs. To demonstrate the utility of this system, I recreate four real-world responsive visualization examples selected from the analyzed visualization corpus. For each example, I provide a step-by-step walkthrough for how to develop the visualization designs. These walkthroughs demonstrate how a user can construct, compare, customize, and iterate on different visualizations using a flexible development workflow that contrasts to the predominantly linear approach described by journalists.

Each of these projects pushes beyond *what* visualization construction systems can produce, to think about *how* users engage with the system or process to reach the desired result. By shifting the system focus to the abstractions and concepts most relevant to users, we can produce an experience that better reflects the user's goals and expectations. Doing so also opens opportunities for iterative changes to the underlying system and the development of smart default functionalities. In other words, since users are not responsible for directly manipulating the system behavior, we can introduce changes to the functionality (e.g., updating the underlying algorithms or constraint solver) without impacting the user's experience. Throughout this dissertation, the main focus remains on how to surface the parts of this underlying functionality that are most essential to the design decisions of the user.

1.2 Thesis Outline

This dissertation contributes new programming languages and visualization techniques for end-user visualization authoring systems. The content of this thesis is organized as follows:

Chapter 2 surveys related work across projects in this thesis, including: visualization design systems, graph visualization techniques, particularly those leveraging constraints, program visualization and debugging, approaches to text and environment augmentation, empirical studies of programmers, and both declarative and domain-specific programming languages.

Chapter 3 describes a set of formative interviews with programmers developing constraint-based applications. Participants describe their development experience and challenges that arise when utilizing constraints, particularly for end-user facing systems.

Chapter 4 contributes SetCoLa [83]: a new high-level language for specifying customized graph layout based on domain-specific properties (e.g., node attributes and topology) using constraints. I show how SetCoLa can support the design of custom layouts with an order of magnitude fewer user-authored constraints than previous approaches, and can support reuse of the layout across graphs from within the same domain.

Chapter 5 explores program understanding techniques in Vega [177]: a declarative visualization grammar for creating interactive visualizations. I first describe the design of a prototype data flow graph visualization of Vega’s underlying system functionality. Chapter 5 further contributes a set of formative interviews with expert Vega users about their development and debugging process, as well as the utility of the data flow graph visualization as a tool for program understanding by end-user programmers of Vega.

Chapter 6 introduces a set of visual debugging techniques for reactive data visualization [87], which are inspired by the formative interviews from Chapter 5. I show how these visualizations can help novice programmers effectively understand source code to identify bugs or crucial dependencies in unfamiliar code written in an unfamiliar programming language.

Chapter 7 contributes a design space of code-embedded program visualizations that aim to unobtrusively reveal runtime behavior during both normal execution and debugging [88]. Leveraging these visualizations for Vega, I show that users can more accurately answer program understanding questions with the aid of inline visualizations of the code behavior.

Chapter 8 explores the design of responsive visualizations: visualizations that adapt based on the screen size or interactive capabilities of the device. I contribute a set of formative interviews with journalists and a survey of visualizations from news articles to explore how responsive techniques are used. I then contribute a system for responsive visualization design that supports simultaneous editing and device-specific customizations [84].

Finally, Chapter 9 summarizes the contributions, takeaways, and techniques employed across each project for addressing the three challenges introduced in this chapter. Chapter 9 further describes the potential impact of this dissertation to future work.

The remainder of this dissertation document includes the bibliography and additional appendices with other relevant material beyond the scope of the individual chapters. Specific appendices are referenced from the relevant chapters throughout the dissertation.

How to Approach This Dissertation

This document includes years of work and is therefore rather long, as dissertations generally are. If you are interested in a particular project, consider reading the original paper for that work. That being said, in some cases the dissertation chapter may include a more thorough or revised version of the publication content. In particular, consider reviewing the additional figures that are included in the dissertation, which are beyond the scope of the originally published papers. Furthermore, not all chapters in this thesis include a corresponding publication and are therefore the only source of information on the topic. For a longer, while still brief, overview of all the work described in this thesis, you may refer to the individual “Summary of Contributions” section in each chapter.

1.3 Prior Publications and Authorship

Throughout this dissertation I present work for which I am the primary author, but I would like to acknowledge that each and every project was done with the help and collaboration of my research mentors. SetCoLa was published at EuroVis 2018 [83] in collaboration with my advisor, Jeffrey Heer, and Alan Borning. For the Vega project, which was published at IEEE VIS 2015 [177], I contributed figures inspired by my work on data flow visualizations, which I presented at a EuroVis 2015 workshop on Reproducibility, Verification, and Validation in Visualization (EuroRV3) [86]. This work on program understanding and debugging for Vega inspired my projects on visual debugging techniques (EuroVis 2016 [87]) and in situ visualizations (ACM CHI 2018 [88]), which were both done in collaboration with Jeffrey Heer and Arvind Satyanarayan. Finally, my work on responsive visualization design was published at ACM CHI 2020 [84] in collaboration with Zhicheng “Leo” Liu and Wilmot Li from Adobe Research. I will reiterate this information on collaboration in the “Summary of Contributions” section for each chapter. When describing my work throughout this dissertation, I will use the first person plural to describe the work of myself and my collaborators.

Chapter 2

BACKGROUND AND RELATED WORK

There are many approaches to visualization design, depending on the ultimate goals of the end user. To this end, I first outline the space of programming languages and tools available for constructing visualizations (Section 2.1), which includes programming languages for visualization design (Section 2.1.1), end-user visualization construction systems (Section 2.1.2), and visualization systems for exploratory data analysis (Section 2.1.3). Graph visualizations can prove particularly difficult to design effectively; Section 2.2 therefore introduces related work on specific techniques for graph visualization, including graph layout techniques that leverage constraints (Section 2.2.1). Continuing the visualization theme, Section 2.3 describes the application of visualizations for program understanding and debugging, including the design of in situ visualization of the program behavior. Related to the design of in situ visualizations, Section 2.4 describes related work on text and environment augmentation.

While the approaches and tools themselves are important, understanding the experience and behavior of the end user is just as essential; Section 2.5 introduces work on empirical studies of programmers and program understanding. Section 2.6 describes particular program understanding techniques explored in the space of declarative programming languages. Finally, Section 2.7 introduces related work on domain-specific programming languages.

To conclude this chapter, Section 2.8 describes the applicability of this related work to the content described throughout this dissertation. Individual sections of the related work in this chapter highlight specific relationships of immediate interest. Furthermore, some chapters in this dissertation include additional related work sections specifically related to the content of that chapter (e.g., Section 3.1, Section 4.1, Section 5.1, and Section 8.1).

2.1 Visualization Design Systems

Visualization construction systems target different goals for the visualization design and different levels of expertise for the user. In exploratory data analysis, users strive to understand the underlying data to uncover analysis insights. Visualizations can then be used for communicative purposes to inform others about important observations and trends. Visualization construction systems enable users to implement a particular visualization design, but such systems often expect users to be familiar with their data and have a particular design in mind.

2.1.1 Programming Approaches for Visualization Design

Programming systems provide the most flexible and customizable approach to visualization design. Languages such as D3 [23], Processing [157], and Vega [177] are particularly popular for designing visualizations programmatically. Whereas D3 supports interactive visualization design via event callbacks, Vega leverages a declarative approach that makes it particularly well-suited for the programmatic generation of visualization specifications. Given this advantage, Vega also serves as the underlying platform for an ecosystem of tools [174, 176, 195, 206], including several of the projects presented in this dissertation [84, 87, 88]. While these programming approaches can support a huge range of designs, they require extensive expertise to learn and employ effectively. Simpler languages such as Vega-Lite [176] can provide a quicker, more lightweight approach to visualization design, but still require textual specification. In this dissertation, we contribute SetCoLa [83]: a domain-specific programming language for customized graph layout using constraints. SetCoLa is discussed in more detail in Chapter 4.

2.1.2 End-User Visualization Construction Systems

An alternative to programming approaches are visualization construction user interfaces. Grammel et al. [69] surveyed such approaches and identified two main interface types: *template editors* and *shelf construction systems*. Satyanarayan et al. [175] revisit this distinction and introduce a third interface type: *visual builders*. As part of this work, Satyanarayan et al.

compare and contrast three systems, which they classify as *visual builders*: Lyra [174], Data Illustrator [129], and Charticulator [161]. Satyanarayan et al. then provide critical reflections on the design of these three systems and make explicit the implicit assumptions adopted in the early implementation of each system, thus surfacing new areas for future work. I will briefly describe these three classes of visualization construction systems here:

Template editors generally support a limited set of visualization designs. While some systems support design customization to modify the visual style, these systems can only support new visualizations through the introduction of new templates. Adding new templates can be difficult and often requires additional programming expertise. Examples of template editors include Datawrapper [64], Flourish [131], RAWGraphs [135], and Microsoft Excel [142].

Shelf construction systems utilize a drag-and-drop mechanism in which users map data fields to encoding channels to produce the visualization design. These systems must often make a number of default decisions in order to produce valid designs based on the specifications of the user. Examples include Tableau (formerly Polaris [184]), and research systems such as Polestar and Voyager [206]. Our proposed responsive visualization design system [84] primarily employs the shelf construction approach to map data fields to encoding channels via drag-and-drop, with minor modifications possible via direct manipulation (Chapter 8).

Visual builders are the most complex interface of the three, but also provide the most control and customization of the design. Lyra [174] provides an interactive environment for visualization design via direct manipulation, which was built on top of Vega [177]. Data Illustrator [129] follows a similar approach, modeled on existing vector graphics programs. Charticulator [161] similarly enables direct manipulation design of visualizations, and additionally leverages constraints to produce the chart layout. Satyanarayan et al. [175] compare and contrast these three systems [129, 161, 174] to provide critical reflections on the different approaches. Other examples of visual builders include Data-Driven Guides [111], VisComposer [138], iVoLVER [139], iVisDesigner [160], InfoNice [203], and DataInk [208].

2.1.3 *Exploratory Analysis and Visualization*

Visualization plays a crucial role in data and statistical analysis by confirming that the data matches the expectations of the user, revealing errors in the underlying data, or surfacing data insights. `ggplot2` [204] is a visualization design language for data analysis in R. Altair [195] is a Python library for lightweight visualization design built on top of Vega-Lite [176]. For these approaches, designers are responsible for individually and programmatically specifying each visualization in the analysis process. As an alternative, Voyager [206] is a mixed-initiative system to support data exploration using both automatic and manual specification of visualizations; the user is presented with a series of different visualizations and can steer which visualizations are shown by partially specifying the design to focus on the data of interest.

2.2 *Graph Visualization Techniques*

Graph layout approaches often leverage the underlying structure to produce the layout [49, 63, 81]. For node-link diagrams of hierarchical data, Reingold & Tilford’s “tidy” layout [158] arranges graph nodes into compact, symmetrical tree layouts based on aesthetic properties. Radial layouts [13, 81] follow similar procedures using polar coordinates, with a root node placed at the origin. Sugiyama-style layouts [186] visualize directed graphs by first assigning nodes to hierarchical layers and then iteratively adjusting node placement to minimize edge crossings. Force-directed techniques [56, 119, 155, 193] use physical simulation and/or optimization methods that model repulsive forces between nodes and spring-like forces on edges, and attempt to minimize the overall energy. Origraph [19] is an end-user system that supports interactive network wrangling to restructure and visualize the underlying network data. A number of popular tools support graph drawing, including D3.js [23], Gephi [11], Graphviz [50], and Cytoscape [181]. This work on graph layout inspires both the design of SetCoLa (Chapter 4) and the prototype data flow graph visualization for Vega (Chapter 5).

2.2.1 *Constraint-Based Graph Visualization Layouts*

Extending an existing layout method to support constraints enables customized layouts that emphasize important structural or aesthetic properties of the graph. There is extensive re-

lated work in this space that explores the application of constraints for different use cases. Dig-CoLa [42] encodes the hierarchy of nodes as constraints and attempts to minimize the overall stress; this technique is a hybrid strategy that combines automatic hierarchical layout with undirected layouts to ensure downward pointing edges. Dwyer and Robertson [47] present a strategy for supporting non-linear constraints (e.g., circle constraints) that does not constrain node positions along a single axis, and can incorporate these techniques into other layout strategies. Kieffer et al. [109] present a force-directed, constraint-based layout for creating graphs with node and edge alignment, and demonstrate its effectiveness for interactive refinement within an end-user facing system for interactive graph layout: Dunnart [45]. IPSep-CoLa [43] extends force-directed layouts to apply separation constraints on pairs of nodes to support properties such as customized node ordering or downward pointing edges. Dwyer and Wybrow developed libcola [48], which utilizes constraints within a force-directed graph layout [46] using stress majorization. Stress majorization [60] is a technique used for graph layout that has been extended for efficient application on constrained layouts [44, 202].

WebCoLa [41] is a JavaScript library based on libcola [48] for constraint-based layout in a web-programming context that can be used alongside D3 [23] or Cytoscape [181]. WebCoLa enables constraints on the alignment and position of nodes as well as the specification of high-level properties such as flow (to ensure edges point in the same direction) and non-overlapping constraints. While WebCoLa can support customized constraints, the specification of individual inter-node constraints can be labor intensive. To address this concern, we developed SetCoLa [83]: a domain-specific language for specifying high-level constraints for graph layout. The SetCoLa compiler (described in Chapter 4) translates the user-authored constraints into low-level constraints for WebCola [41], which computes the final layout.

2.3 Visualizations to Facilitate Program Understanding and Debugging

Program visualization can facilitate both educational and debugging tasks. The Online Python Tutor [71] visualizes objects, variables, and stack frames allowing students to inspect the runtime state of their code. The Online Python Tutor has been used by over 3.5

million people and has been extended to support additional languages including JavaScript and C. Algorithm visualizations [21, 37, 180] can illustrate code by visualizing each step in the algorithm. Such approaches have been shown to improve understanding of the behavior [70].

Many debugging tools provide coordinated views to display relevant system information and facilitate tracing of the execution history. Brad Myers describes a taxonomy of program visualizations—in contrast to visual programming—that are “*used to illustrate some aspect of the program or its run-time execution*” [144]. The Whyline [115] visualizes the path of runtime actions relevant to a “why” or “why not” question about the runtime behavior. Timelapse [26] visualizes web event streams and displays linked views of internal state information; breakpoints allow programmers to trace state information to particular parts of the original source code. FireCrystal [149] emphasizes the connection between code and runtime behavior by extracting the relevant CSS, HTML, and JavaScript code responsible for behaviors on a web page. Theseus [128] similarly narrows the connection between runtime behavior and code by displaying visualizations of program calls alongside the source code and call stack. Omnicode [102] employs a scatterplot matrix to visualize the entire execution history of runtime variables, which helps novice Python programmers debug and explain the behavior of their code. These types of environments with coordinated views have motivated our visual debugging techniques proposed in Chapter 6; in our work, we focus on program understanding techniques for reactive programming languages such as React [93] and Elm [36].

Many of these debugging tools utilize separate coordinated views that require programmers to switch between code authoring and debugging tasks. In situ visualizations aim to narrow the gap between the source code and program understanding techniques. In situ visualizations of program behavior have been developed for a variety of programming topics to show properties such as the edit history or code authorship [73], variable read/write accesses [14], performance behavior [15], and real-time programming tasks [189]. Bret Victor [198] describes design considerations for a programming system to support program understanding tasks aided by both code annotations and visualizations. We build upon much of this related work to contribute a design space of code-embedded visualizations in Chapter 7.

2.4 Text and Environment Augmentation with Visualizations

Related to the in situ visualization techniques described in Section 2.3, text augmentations display additional information to support or extend the text [28, 211]. Tufte [192] describes sparklines, which are small, data-rich visualizations incorporated into text. Goffin et al. [65] have generalized this idea to include any form of word-scale graphics. Goffin et al. [66, 67] additionally present design considerations for the placement of word-scale graphics and for incorporating interactions. Latif et al. [125] present a design space of interactions for linking between text, visualizations, and sparklines in interactive documents. Willett et al.’s scented widgets [205] augment standard navigational widgets with visualizations of page visitation or content engagement. Such visualizations orient users within the space of possible content to view, and help them to engage with underutilized content. This work strongly motivates the design of our in situ visualizations for program understanding in Vega (Chapter 7); in Chapter 7, we contribute a design space of code-embedded visualizations and elaborate on placement considerations particularly for programming contexts.

2.5 Empirical Studies of Programmers and Program Understanding

Program understanding plays an essential role in how programmers learn, debug, and author new code. There is extensive prior work examining how programmers understand code [165, 185, 199]. Ko et al. [117] present six learning barriers in end-user programming systems, which includes: design, selection, coordination, use, understanding, and information barriers. Oleson et al. [148] describe learning difficulties from the perspective of students. Ko & Myers [114] contribute a model of programming errors and discuss the evaluation of this model on two studies of programmers using Alice. Many pieces of related work aim to understand how potential users communicate ideas and concepts while uninhibited by a particular programming language or environment [130, 146, 150, 212].

As described in Section 2.3, many debugging tools utilize separate coordinated views that require programmers to switch between code authoring and debugging tasks. However, requiring programmers to shift their attention between views can incur a high context switching

cost and interrupt programmers' flow when writing code. Flow [147] is a well-studied topic from the field of psychology and describes the state in which an individual "operates at full capacity." While "in flow," programmers may introduce but overlook errors in their code. Saff & Ernst [167] describe a model of developer behavior where the phase in which programmers have unknowingly introduced an error is called "ignorance." Across two development projects, Saff & Ernst [167] found that Java programmers were ignorant of program errors for about 17 minutes on average (and sometimes more than 90 minutes).

Once an error has been introduced, programmers must invest time to find and fix the error, which often requires programmers to switch between code authoring and debugging tasks; Ko et al. [118] found that programmers spend 5% of their development time switching between tasks on average. Ko & Myers [116] found that programmers spend 46% of their time debugging. Saff & Ernst [167] found a significant relationship between the "ignorance time" of an error and the amount of time needed to fix it. Parnin & Rugaber [151] studied the time required to resume programming after an interruption and found that 30% of programming sessions had a lag of over thirty minutes between entering the session and authoring new code, potentially as a result of extensive debugging tasks. This dissertation aims to facilitate code authoring and understanding by emphasizing the domain-specific concepts familiar to users thereby reducing the gap between the code and program output; Chapter 7 contributes a design space of code-embedded visualizations that help surface surprising or incorrect results directly within the code to reduce programmers' need to switch amongst different views.

2.6 Declarative Programming Languages and Debugging

Declarative languages have commonly been explored for visualization [23, 78, 177, 204]. The declarative approach yields benefits, such as supporting retargeting across platforms [78] or facilitating system-level optimizations. This approach can further enable the programmatic generation of visualizations, thus providing a base for the design of new visualization systems [84, 174, 206]. However, the declarative approach also surfaces challenges for effective debugging; in particular, users may struggle to understand how the user-provided input maps

to the system-produced output. Within this space, there remain new opportunities to better support the program understanding and debugging process. In this dissertation, Chapters 5-7 contribute new debugging techniques for reactive visualization in Vega [177]. Similar to our debugging techniques, the Elm language [36] has explored the development of an interactive debugging environment for reactive programming, inspired by Bret Victor [197].

2.7 Domain-Specific Programming Languages

Domain-specific languages (DSLs) are programming languages that are customized to reflect the knowledge and constructs of a particular domain, such as web computing [3, 27, 51, 58]. Van Deursen et al. [194] describe a survey of 75 publications exploring the use of domain-specific languages for software systems. DSLs have also been proposed for a variety of visualization use cases, which includes designing mathematical diagrams [209], analyzing and processing images [112, 156], as well as graph analysis [89] and layout [83], among many others. For example, in this dissertation we present SetCoLa [83], which is a domain-specific language for customized graph layout that allows users to map their knowledge of the underlying graph properties to visual requirements in the layout using constraints (Chapter 4).

Extensive prior work has described the design of domain-specific languages [53, 91, 140], including analyses of common design patterns [152, 183]. Compared to general-purpose programming languages, domain-specific languages raise the level of abstraction to help users reason about program behavior using the high-level constructs of the domain of interest. However, it can be challenging to develop effective DSLs. To this end, prior work has explored domain-agnostic ways to support the development of domain-specific languages [12, 123, 163].

Domain-specific languages can usefully abstract details of the implementation by enabling users to focus on high-level properties of the particular domain. To illustrate the benefits of DSLs for program comprehension, Kosar et al. [122] present three studies comparing domain-specific languages to general purpose languages and show that developers are both more accurate and more efficient when working with DSLs. In recent work, Kosar et al. [121] replicated these results to further demonstrate the advantages of DSLs for program comprehension

when used within an integrated development environment (IDE). While DSLs have many advantages, users can encounter unique challenges when trying to understand the behavior of internal errors that are obfuscated by the high-level language. Furthermore, these languages often lack debugging tools. This dissertation first explores the challenges around developing a DSL for graph layout using constraints (Chapter 3-4) and further contributes new techniques for program understanding of interactive visualizations in Vega (Chapter 5-7).

2.8 Discussion and Applicability of Related Work

This dissertation primarily explores the design and utility of systems for interactive visualization design; therefore, this dissertation is first motivated by related work on visualization construction systems (Section 2.1). Inspired by work on programming languages for visualization design (Section 2.1.1), graph layout (Section 2.2), and domain-specific languages (Section 2.7), this dissertation contributes SetCoLa [83]: a new domain-specific programming language for designing customized graph layouts using constraints (Chapter 4).

Chapter 5 includes additional background on Vega (Section 2.1.1), and contributes a prototype data flow graph visualization inspired by related work on graph layout (Section 2.2) and program visualization and debugging (Section 2.3). Chapter 6 contributes a set of visual debugging techniques for reactive data visualization inspired by program understanding environments (Section 2.3) and empirical studies of programmers (Section 2.5). To better address the remaining program understanding challenges surfaced in Chapter 6, Chapter 7 contributes in situ visualizations likewise inspired by related work on visualizing program behavior (Section 2.3) as well as work on text and environment augmentations (Section 2.4).

Finally, Chapter 8 describes the design of a new visualization construction system targeted at the design of responsive visualizations. Unlike prior visualization construction approaches (Section 2.1), our responsive visualization system supports the concurrent design and customization of multiple linked visualizations. This environment is further motivated by prior work on text and environment augmentations (Section 2.4).

Chapter 3

UNDERSTANDING THE PROGRAM BEHAVIOR OF CONSTRAINT SYSTEMS

The first core challenge presented in this dissertation is to “raise the level of abstraction to reflect user expertise.” One way in which to encode user expertise is through the application of constraints. Constraints are a flexible and powerful approach to solving complex problems. By applying constraints, users can express behaviors or circumstances of immediate relevance to their unique domain-specific knowledge and their primary task at hand. For example, constraints can be particularly applicable to visual layout use cases including visualization, graph layout, or interface design. For these tasks, constraints provide a natural way in which to encode requirements for the layout such as alignment, element order, and hierarchy, while also providing some flexibility to the design. Constraints can also be applied for more abstract use cases such as scheduling [54], automating statistical analysis procedures [99], and the compilation and reapplication of visualization design guidelines [143]. These approaches provide an extensible way to encode domain knowledge, which can better support reuse and reapplication of this expertise. Essentially, constraints are able to “support the tasks that matter most to the user” by allowing users to encode their knowledge while introducing additional flexibility in the system functionality.

While constraints exhibit the potential to support users in solving complex problems by enabling them to encode their unique domain expertise, the application and invalidation of constraints sometimes remains hard to comprehend. Furthermore, while constraints may naturally reflect some domain knowledge (such as alignment requirements), users sometimes struggle to identify how best to author or prioritize constraints for their particular task. To further complicate this process, techniques to support program understanding and de-

bugging of constraint systems is currently limited. The difficulty surrounding the program understanding of constraints illustrates the second challenge explored in this dissertation: “communicate system behavior as actionable information.” In order to develop new systems to support users in understanding the behavior of constraints, it is important to first identify the challenges and opportunities surrounding existing constraint programming systems. To this end, inspired by prior work on empirical studies of programmers and program understanding (Section 2.5), this chapter aims to uncover existing strategies and challenges surrounding program understanding of constraints, and to identify new opportunities for future research. By better supporting program understanding in complex domains such as constraint programming, we can open the domain to a wider group of end-user programmers and better support the effective use of new constraint-based systems.

3.1 Related Work: Constraint Programming Systems

Constraints can help solve complex problems including user interface design [2, 38, 187, 188], visualization layouts for graphs [41, 43, 45, 48, 83], charts [161], and diagrams [20, 172, 209], the compilation and reapplication of visualization design guidelines [143], scheduling [54], and the automation of statistical analysis procedures [99]. Constraints can simplify complex problems by focusing the user’s attention on design essentials (such as the interplay amongst related or disparate elements in the constraint problem). Several of the systems introduced in this section take the abstraction one step further to produce a graphical user interface that hides the use of constraints in the underlying system [45, 161, 187]. Michael Sannella has explored debugging techniques for constraints in interactive user interfaces [169]. This section focuses primarily on examples of end-user facing systems that leverage constraints.

As previously discussed in Section 2.2, constraints have been thoroughly explored within the graph layout domain. Related to this body of work is a graphical user interface called Dunnart [45], which is a graph layout environment that supports interactive refinement of the layout powered by a force-directed, constraint layout engine [109]. Another example of a system for graph layout using constraints is SetCoLa [83], which is discussed in more detail in Chapter 4. In SetCoLa, the system generates constraints based on a specification provided

by the user to produce a customized graph layout. The SetCoLa compiler generates constraints for a low-level constraint engine for graph layout called WebCoLa [41], and displays the system generated WebCoLa constraints using a similar format to the initial SetCoLa specification. However, the number of constraints in the generated specification can increase by one to two orders of magnitude, which complicates how users might go about understanding the behavior of the final graph layout. As I will discuss later in this chapter, a common challenge across different constraint approaches is that constraints can be tightly coupled such that changes to one part of the solution cascade throughout the constraint problem.

While constraints have been extensively studied for graph layout, the flexibility of this approach also applies to other visualization contexts. Charticulator [161] is an example of a visualization construction system (Section 2.1.2) for producing bespoke chart layouts powered by constraints. Charticulator aims to support the design of visualizations without programming. To do this, Charticulator includes a set of user interactions for specifying properties of the layout that are represented as constraints in the underlying system. The system then uses a least squares minimization technique for weighted constraints to produce a layout. While this solver is faster for the particular types of constraints generated by Charticulator, it does limit the types of constraints that can be defined to equality constraints. The constraints produced by Charticulator are entirely hidden from the user, such that it can be hard for the user to understand what happens when the produced layout differs from the user’s expectations. Beyond the visualization domain, ThingLab [20] is a graphical programming environment that allows users to interactively create physical simulations using constraints.

There are several other end-user systems that leverage constraints for the underlying layout of graphical interfaces. Rewire [187] is a system to support designers in prototyping user interfaces, which leverages constraints to ensure alignment and consistency in the user’s design. Scout [188] is a system for generating user interface design variation based on user-authored constraints. Constraint approaches for interface design are particularly common within industry, including Apple’s AutoLayout [2] which leverages an incremental constraint-solving algorithm called Cassowary [4], and Android’s ConstraintLayout [38].

3.2 Formative Interviews: Utilizing and Understanding Constraints

To better understand existing challenges and inform the design of new program understanding techniques for constraint systems, we conducted preliminary formative interviews with end-user programmers with experience designing and using constraint programming systems. The goal of these interviews is to surface potential challenges and suggest new avenues for future work on techniques for program understanding in constraint systems. This section describes a preliminary set of interviews with four participants to highlight some common challenges that arise when working with constraints. Future work can leverage this methodology and preliminary set of insights to inform the design of new studies on program understanding for constraints and new interventions to improve user interaction with constraints, for both system developers using constraints and potential end users of constraint-based systems.

Participants. We recruited four participants (2 male, 2 female) from the University of Washington. All participants were PhD students with prior experience using constraints for personal and/or research projects. For the projects discussed in the interviews, participants used solvers such as Z3, CBC, and Clingo. Participant ages ranged from 24 to 31 (mean 26.25, s.d. 3.20). Each participant received a \$20 gift card for completing a one hour session.

Data Collection. The interviews were conducted at the University of Washington. We captured audio recordings and took notes during the interviews for later review. We then used Rev [162] to transcribe the audio recordings after the interviews.

Protocol. Prior to the interviews, participants completed a background survey in which they briefly described their previous experience working with constraints and provided other demographic information (see Appendix A.1). For the interviews, we utilized a semi-structured approach following a basic interview question template (see Appendix A.2). To inform the structure of these interviews, we conducted two pilot surveys with a similar set of questions.

Analysis. To analyze the results of these interviews, we reviewed the interview transcripts and used an affinity diagramming approach to group participant quotes based on the topics and ideas discussed. We further distilled these groups to identify a set of high-level topics and challenges which we discuss in more detail in the following sections.

3.2.1 *Natural Representations of Domain Expertise*

The decision to use constraint-based approaches arises from the perceived appropriateness or ease with which the domain reflects the nature of constraints. One participant explained a straightforward approach to apply domain knowledge when generating constraints: “[We were] basically just kind of putting numbers to things that we already kind of knew what the priorities were” (P1). While participants would often consider using other approaches to solve the problem of interest, one participant explained that “It was easier for me to declare a [property] as following from a set of constraints about the data, then to try to navigate a pretty dense if-else tree” (P4). In these cases, participants felt that the constraint structure could concisely represent important domain knowledge that would otherwise be hard to navigate with conventional programming approaches. Another participant explained that “it’s more about... how we make the representation more natural, so that other people can easily compile their thoughts and knowledge into the constraint system” (P3).

In the quote above, P3 noted the importance of intuitive constraint representations to foster external collaborations. Constraints can also introduce flexibility and power into end-user facing systems, but only if such systems can support the end user’s development needs. One participant noted that “all the constraints I made, I’m trying to base them off design principles that I found. I’ve been doing a lot of research on different things that designers would wanna do and I’ve been trying to make [the constraints] naturally fit into the types of concepts [designers] would use” (P2). Given that constraints may naturally reflect the end-user’s domain expertise, systems can abstract the implementation to hide those details from the end user. One participant reflected that “I feel like interface designers don’t necessarily want to think in that way. So I’m not even telling them they’re constraints anymore” (P2).

3.2.2 Challenges Formalizing Domain Knowledge

While some domain expertise may naturally reflect the structure of constraints as we saw in the previous section, other types of domain knowledge may prove challenging to formalize. As one participant explained, *“it can be challenging to encode the different rules into the constraints that the software will understand”* (P2). One of the first steps for specifying a constraint problem is identifying the individual statements; P2 further reflected that *“I think just breaking [the problem] down into different statements... is really hard”* (P2). To fully specify a constraint problem requires the user to write a variety of unique, yet interconnected constraints. For many debugging scenarios it can be hard to navigate and iterate on a constraint problem that is underconstrained or overconstrained. One participant explained that *“It’s easier to add a new rule. But with a new rule, it’s hard to tune to work with other rules”* (P3). Another participant felt that *“Removing them is harder, I think, because if you have it too constrained it’s hard to figure out without seeing more examples. Because if it’s just one example... it looks right. But maybe you’ve constrained it too much and there won’t be enough variation and that’s harder to debug, I think”* (P2).

Producing the desired result often required participants to change and iterate on the set of constraints. One participant explained that this process could be difficult because *“I had a hard time evaluating what I had already done and what I hadn’t tried and which one was better or worse”* (P4). When iterating on a large set of constraints, it becomes essential to recall the behavior, rationale, and interplay for different parts of the code. One participant noted that *“sometimes I go back to it later and I’m like, ‘I don’t need this constraint’ and then I’ll remove it. But after I remove it... I ran it and it was really slow. And I’m like, ‘Oh, that’s why I did that’”* (P2). While many participants primarily worked on the code by themselves, collaboration further emphasized the importance of documenting the rationale. However, one participant noted that *“My original code may have some explanation, but it’s just not something that other people, whoever comes in... it’s really hard to continue working on that. How to make the code self-explanatory is also a very challenging problem”* (P3).

3.2.3 Requirements and Challenges Around Solver Performance

A major challenge for our participants was around understanding the performance of the underlying constraint solver: *“performance was more important, ‘cause we were calling the solver a lot, so we were really thinking about like... ‘how can we avoid calling the solver? Can you optimize our queries?’”* (P1). One participant commented on the general difficulty surrounding performance to note that with *“the performance issue... it’s really, really hard to understand what’s wrong here”* (P3). Another participant reflected that *“it’s just with the way you write them it can make the software go slower or fast, I don’t really know why, it’s just how they work”* (P2). When it comes to understanding the overall behavior of the application, it can be particularly hard to identify program understanding strategies that reflect both the user-authored constraints and the behavior of the underlying solver. One participant went on to explain that *“the solver, that part was kind of a black box that is something I cannot control and I don’t know what’s going on”* (P3). As one participant explained, overcoming a performance issue can often require extensive time and/or user effort: *“I had to redo all of my constraints so they would be faster”* (P2). While participants may be able to produce the desired behavior, it still remains difficult to know whether additional improvements are possible: *“I want to know if there’s a better way of debugging constraints with Z3, because I think what I did worked for my purpose but wasn’t necessarily efficient”* (P4).

3.2.4 Current Approaches to Constraint Debugging

While powerful, constraints present a variety of challenges, particularly around debugging. As one participant stated: *“I think the biggest challenge would be... well there’s a couple. There’s lots of different challenges, so don’t know if there’s one biggest one”* (P2). Another participant explained that *“the first thing is actually debugging, so this constraint system is declarative. It’s very hard to tell which part is wrong”* (P3). This tension is one that we continue to see across other projects in the declarative space, as we will discuss later in this dissertation (Chapter 4-7). Another participant mentioned that *“when it’s working great, it works great, but when it goes wrong, there’s no real feedback”* (P1).

When constraints fail to produce the desired output, or are completely unsatisfiable, the authors must begin to debug the behavior. For some constraint applications, there are basic techniques to help understand the behavior. One participant explained that *“the solver actually has some way to interpret what’s wrong and what’s a conflict just using unsat core, for example”* (P3). However, another participant noted that *“But sometimes [the unsat core] has extra things in there that don’t really make sense, so it’s not easy to interpret that”* (P2). To actually begin making changes, many participants described just commenting out individual lines of code. P2 went on to explain that *“I would use [the unsat core] to figure out which lines to comment out... and if I comment them out and it’s satisfied, then I’ll have to figure out what the issue is with that constraint”* (P2). Other participants explained that the easiest way to understand the behavior was to simply inspect the output: *“When there is something wrong, the only way you can do is to work through all of my codes to inspect what’s really wrong”* (P3). Another participant reflected that *“I just kind of lucked into it, in terms of debugging. I was looking at this line, I was like, ‘Wait a minute, that’s not right.’ So, really, no useful output from the program at all, it was just kind of like inspection of the code, which is unfortunate, but it is what it is”* (P1).

3.2.5 Takeaways: Understanding and Appropriately Employing Constraint Systems

For some domains—such as graph or interface layout and scheduling—constraints can provide a natural way to record and apply domain knowledge to programming problems. However, challenges may also arise when determining how best to break the particular problem into small enough steps to formalize in both system and human understandable language. Collaboration can prove essential for the collection and reapplication of domain knowledge as constraints; however, collaboration also introduces requirements for improving how to communicate the behavior of disparate elements in the constraint problem. For end-user facing systems, it is particularly important for constraints to reflect the goals of the end user at a sufficiently high-level to abstract away the underlying constraint implementation.

Many challenges arise when it comes to understanding the behavior of a constraint-based system. Understanding the interplay of constraints can be essential when iterating on how best to represent a constraint problem. Challenges often arise for both overconstrained or underconstrained constraint problems requiring the user to reason about the connections (or lack thereof) between constraints. When adding or removing constraints, it becomes essential to document the behavior and rationale so as to avoid repeated effort. In addition to challenges surrounding the program understanding process for individual constraints, reasoning about the performance of the underlying solver can prove particularly challenging. When it comes to debugging, users generally lack the tools necessary to fully explore or interpret the program behavior. New techniques for program understanding of constraints should aim to better support system behavior by leveraging the domain expertise of the user to communicate actionable information and better support the user's debugging process.

3.3 Limitations and Future Work

This chapter presents a preliminary set of interviews with four participants to better understand how individuals leverage, understand, and debug constraints. In particular, these interviews focus on the use of constraints in collaborative settings and in the development of end-user facing systems. While these interviews surface some themes about the existing challenges faced by developers, future work should elaborate on the interviews to recruit a broader class of participants. In particular, future work should aim to understand the experience of end users of constraint-based systems to further identify challenges that arise when working with constraints indirectly and opportunities for new interventions to help end users interpret the complex functionality presented by such systems. The interview methodology described in this chapter can be extended or refined based on these results to help elicit new sets of challenges and to better explore the design of new program understanding techniques.

3.4 Summary of Contributions

Constraints can provide a natural way to encode domain knowledge in the behavior of a system to better support flexibility and prioritization of the underlying behavior. During the interviews, one participant reflected on the power provided by the constraint-based approach, noting that *“we wrote down we valued these things highly, and I guess we expected that we had always valued them highly, and that humans were doing a good job maximizing that. And that turned out to just not be the case, so the tool was really able to outperform in that context, and I think that was surprising”* (P1). However, the process of authoring and understanding the constraint behavior can often be challenging and labor intensive. In this preliminary set of interviews, participants explained a variety of program understanding challenges including the difficulty of formalizing knowledge as individual constraints, reasoning about the behavior of underconstrained or overconstrained specifications, and interpreting the performance and behavior of the underlying constraint solver. Current techniques to debug the behavior generally involves manual inspection of the output or ad hoc experimentation with the constraint specification. These results hint at the extent to which users are underserved by the constraint approaches they currently use.

Many of the challenges discussed in this chapter reinforce the three core challenges surrounding this dissertation, particularly regarding the need for systems to better communicate the underlying behavior and to support the user’s primary tasks. However, these interviews also suggest strategies to help raise the level of abstraction in end-user facing systems. As one participant explained: *“I’ve been trying to make [the constraints] naturally fit into the types of concepts [designers] would use”* (P2). Given this careful consideration, P2 further reflected that *“I’m not even telling them they’re constraints anymore.”* In order to better support the end-user programmer, systems should adapt to the particular expertise of the user rather than requiring users to adapt to the capabilities of the system.

This work was done in collaboration with Jeffrey Heer and Alan Borning. The results of these interviews have not otherwise been published apart from their inclusion in this dissertation.

Chapter 4

AUTHORING AND REUSING DOMAIN-SPECIFIC GRAPH LAYOUTS WITH SETCOLA

Graph visualization research has extensively explored the application of constraints for graph layout (Section 2.2). Constraints can provide a natural way to encode properties of the data to enforce visual considerations such as alignment, node ordering, and hierarchical relationships (Chapter 3). By using an appropriate graph layout, node-link diagrams can effectively convey properties of the network structure, such as the overall hierarchy or network connectedness. Graph visualizations have been commonly explored for analysis and communication in a many domains, including social networks [52, 57, 137, 164, 179], biological systems [6, 16, 61, 120, 127, 170, 181], and ecological networks [7, 18, 32, 82, 103, 124, 126, 166, 210], among others. In many of these examples, the graph layout utilizes domain-specific properties to emphasize relevant patterns in the data. In a biological pathway for example, nodes can be layered by their subcellular location to contextualize the data with respect to the underlying cellular structure (Figure 4.1). The “*transcriptionally regulated genes*” layer of this layout further shows the biological outcomes of this network, grouped by molecular function.

To produce domain-specific graph layouts using existing constraint techniques often requires users to define constraints on individual nodes or node pairs. This process can be labor intensive, requiring thousands of similar constraints and careful reasoning about which nodes should be constrained to produce the desired layout. Moreover, instance-level constraints (e.g., defined *extensionally* via node indices) prevent reuse of a layout across graphs from the same domain and instead require users to author new sets of constraints for each graph individually. Authoring thousands of constraints or automating the generation of constraints may require programming expertise beyond the individual skills of the domain expert.

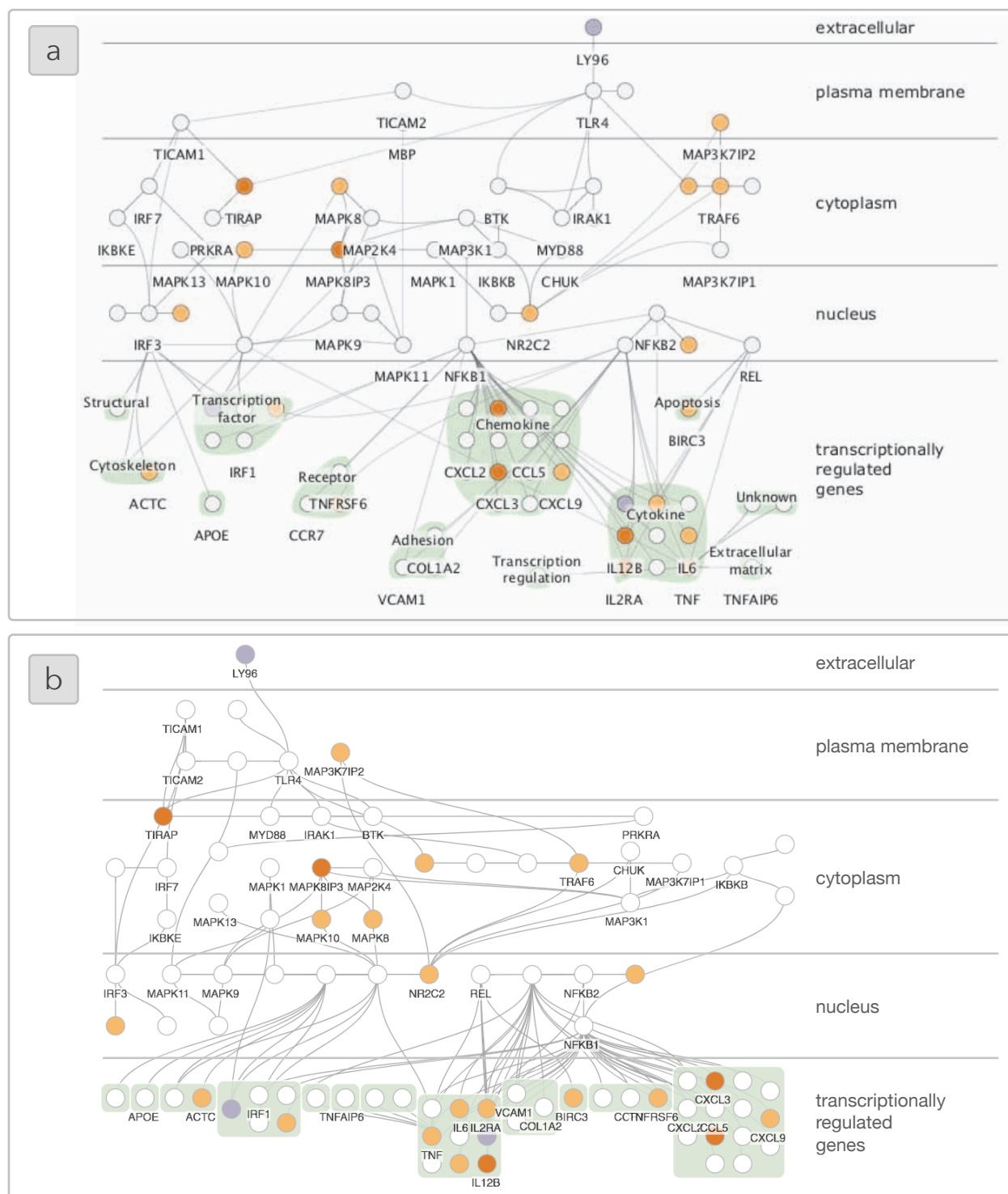


Figure 4.1: The TLR4 biological system layout produced using (a) a domain-specific layout tool, Cerebral [6], as compared to (b) SetCoLa. Layers correspond to the location of the biomolecule within a cell and show immune response outcomes at the bottom, grouped by molecular function.

While constraints provide one option for customized graph layout, many other techniques have been explored to address this need [6, 104, 105, 181]. These techniques leverage common structural properties relevant to the domain—such as known data hierarchies including cellular structure or trophic level—as guiding properties of the underlying layout algorithm. However, these techniques rarely generalize beyond the domain for which they were specifically designed. Furthermore, many other domains lack customized layout tools despite their potential utility. When a layout technique does not exist for the domain of interest, domain experts must either fit their data to available techniques or design and implement a new algorithm. Creating a customized layout algorithm requires both domain and programming expertise, and so introduces a gap between analysis needs and the techniques available. This difficulty helps concretely illustrate the third challenge explored in this dissertation: “support the tasks that matter most to the user.” To produce customized graph layouts, domain experts must be able to leverage their domain expertise to encode relevant layout properties with reduced programming effort. Domain experts should be able to reapply this knowledge across graphs from the same domain to further reduce specification effort for the layout.

To this end, we contribute SetCoLa: a domain-specific language for specifying high-level constraints for graph layout. Users partition nodes into sets based on node or graph properties, and apply layout constraints to these sets. This approach allows domain experts to specify layout requirements at a high level, deferring the generation of instance-level constraints to the underlying runtime system. These SetCoLa constraint definitions reduce specification effort while enabling highly customized and reusable graph layouts. We implemented a SetCoLa compiler, which generates instance-level constraints for WebCoLa [41], a JavaScript library for constraint-based graph layout. To demonstrate the expressiveness of SetCoLa, we recreated several customized layouts found in the scientific research literature [6, 7, 164]. We show that SetCoLa supports compact specification of complex layouts, the output of which resemble those produced by custom layout engines. Our SetCoLa specifications reduce the number of user-authored constraints by one to two orders of magnitude. We also show that these specifications can be reused across different graphs from the same domain.

4.1 *Related Work: Domain-Specific Graph Visualization*

Several techniques have been specifically developed to reflect domain-specific concerns within graph layouts. However, these techniques tend to be highly-specialized, and so may not apply to other possible domains of interest. In other words, the visualizations are often created using specially designed tools or layout algorithms to leverage properties of the data specific to the domain of interest. This section describes examples for ecological networks [7, 106, 107], biological systems [6, 62, 109, 181, 182], and social networks [52, 57, 137, 164, 179].

4.1.1 *Ecological Networks*

Ecological networks are a common visualization to show the relationships amongst organisms in an ecosystem. Baskerville et al. produced a customized visualization of the Serengeti food web in which the nodes are positioned based on their trophic level (e.g., the role of the organism within the larger food chain) and further grouped based on a Bayesian classification of the elements [7]; in addition to the static visualization, Baskerville et al. published an interactive version of the graph online [8]. Despite frequently publishing visualizations of oceanic food webs [106, 107], Kelly Kearney describes several challenges around the design of such visualizations in a blog post [103]; Kearney notes that the node placement algorithm should “*allow constraining y-position to match trophic level while allowing free movement in the x-direction. With no such algorithm seemingly readily available, I decided to create my own.*” In response, Kearney developed customized plugins for D3 [104] and Ecopath [105].

4.1.2 *Biological Systems*

Biological systems also benefit from customized visualizations, such as those produced using Cerebral [6]. Cytoscape [181] is a visualization system designed to explore biomolecular interaction networks and provides a framework for customized plugins, including a WebCoLa plugin for constraint-based layouts. Genc and Dogrusoz [62] describe a constrained force-directed layout technique for visualizing biological pathways. CrowdLayout [182] introduces

a strategy for crowdsourcing biological network layouts from novices; Singh et al. [182] argue that this crowdsourced approach produces more high-quality layouts than Cerebral or Graphviz. Kieffer et al.’s work on incremental grid layouts [109] was motivated by related work for grid layouts of biological networks [6, 120, 127], but aims to provide a more flexible mechanism for creating the constraints by supporting SBGN (Systems Biology Graphical Notation). In later work, Kieffer et al. [110] improve upon grid layout techniques by first identifying the aesthetic criteria humans use for manual graph layout, then producing a new algorithm named HOLA, which employs these techniques for improved, human-like layouts.

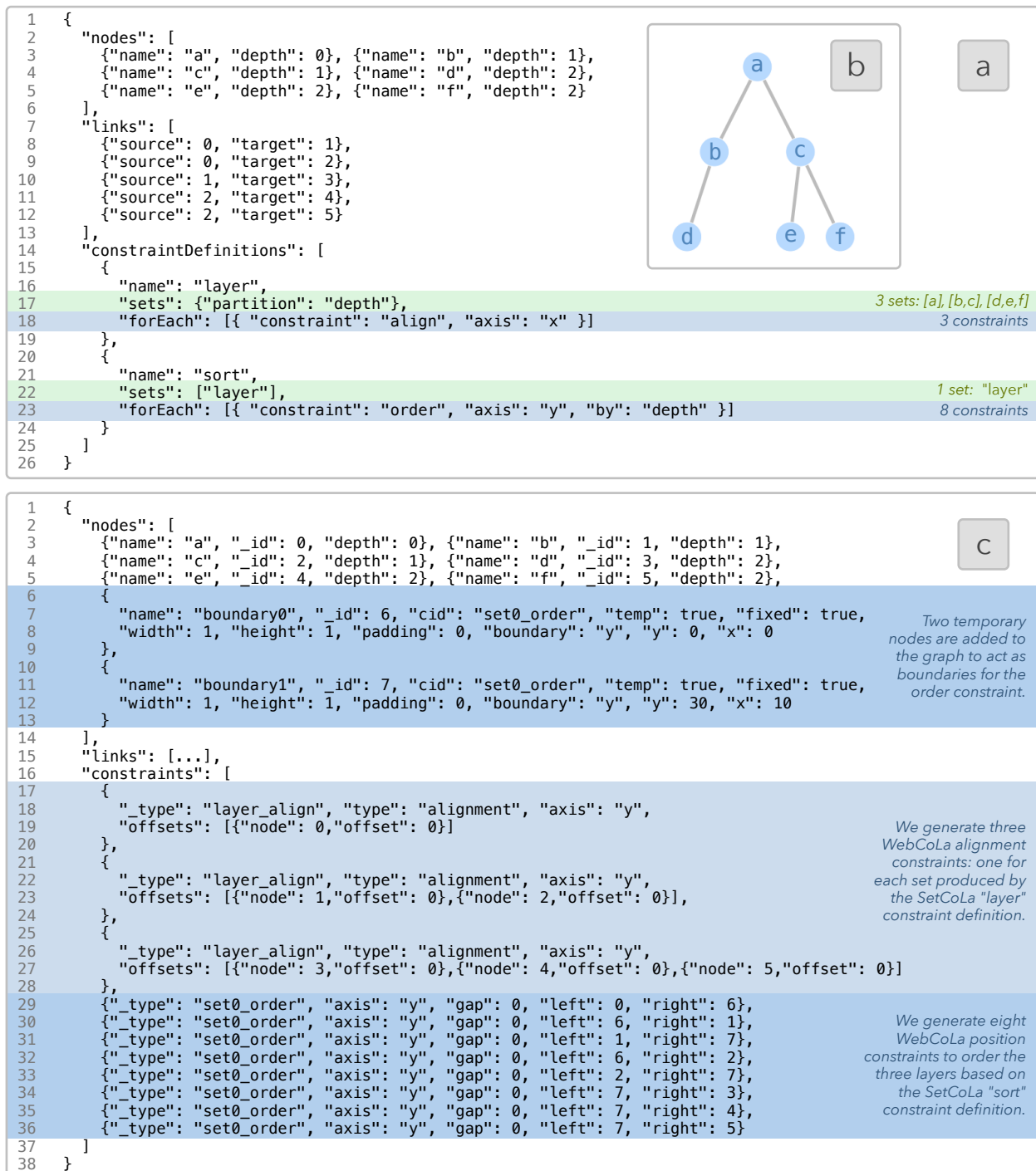
4.1.3 Social Networks

Social networks often leverage force-directed layout techniques to demonstrate the connectiveness or clustering of the graph nodes [179]. However, some network layouts may introduce additional separation or clustering to highlight properties specific to the social network, such as ethnographically-identified groups [164], the timeline of disease exposure [52, 137], or differences in reported relationship types [57].

4.2 Design of SetCoLa: A Set-Based Constraint Layout for Graphs

SetCoLa is a domain-specific language for concisely specifying graph layouts using constraints. To provide a reusable specification without explicit reference to individual nodes and edges, SetCoLa applies constraints to groups of nodes defined by shared attributes. The central abstraction in SetCoLa is a **set**. The simplest elements of a set are graph nodes; however, SetCoLa also supports hierarchical composition, with nested sets as elements.

A SetCoLa specification consists of one or more **constraint definitions**, and an optional set of guides (reference elements that serve as positional anchors). An example SetCoLa specification for a small tree layout is shown in Figure 4.2a. Each constraint definition includes a **set definition** and **constraint application**. SetCoLa provide several operators for defining sets based on node attributes or structural properties, including (1) *partitioning* nodes into disjoint sets, (2) specifying (potentially overlapping) sets via *predicates*, (3) *collecting* nodes



into sets based on key expressions, and (4) *composing* previously defined sets to produce hierarchical layouts. The result of a set definition produces one or more sets, which may have either distinct or overlapping elements. Each constraint definition can define one or more constraints (e.g., for position, ordering, or alignment), which are applied to the nodes within each set created by the corresponding set definition. In other words, constraints defined by a particular constraint definition with multiple corresponding sets are applied to nodes *within* each individual set, not *between* the nodes in different sets.

The SetCoLa compiler takes an input graph and SetCoLa specification, and produces a set of instance-level constraints for an existing constraint-based graph layout solver. In this work, we target WebCoLa [41], a JavaScript library for interactive, web-based layouts. The SetCoLa compiler generates one or more WebCoLa constraints for each SetCoLa constraint to produce a WebCoLa specification (Figure 4.2c). Our implementation of SetCoLa is available at the link: <https://github.com/uwdata/setcola>. In the following sections, we discuss the design of SetCoLa including the process for specifying sets, the types of constraints currently supported in SetCoLa, and how such constraints are applied over node sets.

4.2.1 Specifying Sets in SetCoLa

We provide several operators for defining sets based on node attributes or structural properties, which includes: (1) *partitioning* nodes into disjoint sets, (2) specifying (potentially overlapping) sets via *predicates*, (3) *collecting* nodes into sets based on key expressions, and (4) *composing* previously defined sets. Each of these set definitions produces one or more sets for the constraint definition. For partitions, predicates, or collections, the constraint definition may designate a set from which the elements should be drawn; the default is simply all graph nodes. For each subsection, we show a sample SetCoLa constraint in the header.

Partitioning Nodes into Sets

EXAMPLE: { "partition": "depth" }

The partition operator creates a collection of disjoint sets based on properties of the node (e.g., Figure 4.2a, Line 17). Given n nodes to partition, this operator can produce at most

n sets. The user may reduce the number of sets produced by providing specific property values to **include** or **exclude**. For each node, we identify a key based on the node values for the partition properties and create sets based on the key. The **include** parameter allows the user to identify particular node values to look for when partitioning and includes only those values; the **exclude** property does the opposite. For example, in Figure 4.5, Line 9, the user partitions the nodes by the **group** property and ignores the nodes for which the value of **group** is “*other*.” This partition will produce three sets with the nodes separated by **group**, and the thirteen nodes with group “*other*” will not appear in any of these sets. Any constraints defined in the corresponding constraint definition will therefore only apply to the partitioned nodes and not to the nodes with group “*other*.”

Rationale. The partition operator addresses challenge one of this dissertation—which is to “raise the level of abstraction to reflect user expertise”—by allowing users to employ their domain expertise to recognize and reason over similar groups of nodes. In particular, nodes often include domain-specific properties about different groups in the underlying data, such as the trophic level or location in the cellular structure. Effective domain-specific layouts should therefore encode properties relative to these different groups. Rather than requiring users to reason over the nodes as abstract visual elements tied to data, using this partitioning strategy users can instead encode the data properties directly into the layout.

Specifying Sets with Predicates

EXAMPLE: { "expr": "node.depth == 2" }

For more flexibility in the definition of sets, users can specify a concrete list of sets, each defined by an arbitrary boolean expression. For each expression, each node is evaluated to determine if it should be included in the set. The user may refer to properties of the node using dot syntax; for example, **node.depth** refers to the depth property of the node. In Figure 4.5, Lines 17, 18, and 19, we define three sets based on the **group** property. The set on Line 18 includes nodes that are in the group “*younger white women*” and nodes in the group “*other*.” Users may optionally specify a **name** for the set, which may be referred to in

subsequent composite set definitions. With this specification strategy, it is possible to create node sets that are not disjoint, and may thus lead to unsatisfiable constraints. However, there are also benefits to this flexibility. For example, nodes that occur in multiple sets can act as bridges between different parts of the layout and highlight potential similarities between the different sets that would not otherwise be captured by the simpler partition operator. Given the potential benefits of this flexibility, we chose not to prevent such specifications, though future work should consider adding a warning when overlapping sets have been specified.

Rationale. The predicate set specification strategy aims to support more flexible construction of individual sets than the partition operator. By employing this approach, users can produce a single set containing nodes of different types, produce sets based on combinations of domain-specific properties, and produce overlapping sets to encode more complex relationships in the underlying data. While the partition operator helps raise the level of abstraction to allow users to encode individual domain-specific properties, the predicate operator provides more flexibility. By specifying individual predicates, users can employ their domain knowledge to extract more complex relationships from the underlying data.

Collecting Nodes Using Keys EXAMPLE: { "collect": ["node._id", "node.neighbors.extract('_id')"] }

To combine the flexibility of *predicates* with the automation of *partition*, users may specify sets as a union based on key expressions. For each element on which the constraint definition is applied, each key expression is evaluated to identify the nodes in that set. For example, in Figure 4.6, Line 37, the user creates a constraint definition that applies only to nodes with type “*unknown*” (in this case, only one node). On Line 38, the user creates a set definition with two key expressions: one key expression identifies the `_id` of the node (e.g., `node._id`), and the other key expression identifies the `_ids` of the neighboring nodes (e.g., `node.neighbors.extract("_id")`). The one set produced contains the element itself and all its neighbors. We also include several built-in properties for identifying structural relationships in the graph, which are described in Section 4.2.2.

Rationale. For this set specification strategy, users can programmatically generate complex sets based on structural properties of the graph. Unlike prior work, this approach uses the unique `_id` of the nodes for the layout, while still supporting reuse of the layout across different graphs since the particular `_ids` are only identified during compilation. As with the other set specification strategies, this approach raises the level of abstraction to focus on general structural or domain-specific properties rather than the individual graph nodes.

Composing Previously Defined Sets

EXAMPLE: ["set1", "set2"]

Sets may also be defined as hierarchical compositions of previously defined sets. For example, in Figure 4.2a, Line 17, the first constraint definition (named “layer”) produces three sets via partition. The next constraint definition uses the composition operator, referencing only the “layer” set (Line 22): the result is a single set that contains the three layer sets as elements. For composition, users may refer to any named entities previously defined in the specification (e.g., previous set definitions or named sets produced from *predicates*). In the current version of SetCoLa, we only support composition via set union, though future work should explore the types of set definitions that other composition strategies could enable.

Rationale. While the previous three set specification strategies enable users to encode domain-specific properties in the layout, the goal of the composition strategy is to support hierarchical layouts via nesting. This approach further helps to raise the level of abstraction to represent more complex layouts while employing a consistent specification strategy. In other words, constraints can be applied in the same way as the three previous strategies, but with the base elements representing nested elements (e.g., “sets”) rather than individual nodes.

4.2.2 Built-In Properties of the Graph Structure

In addition to defining constraints relative to node properties, it may also be important to use properties of the graph structure. We support this functionality with a number of built-in accessors. These properties are automatically computed and added to the graph specifica-

tion only when they are used in one of the SetCoLa constraints. These properties are only computed if such a property does not already exist on the nodes and are subject to a number of expectations regarding the graph input; for graphs that do not meet these expectations, users are shown a warning and required to precompute the properties themselves. In this section, we describe each built-in property and discuss the expectations for use.

_id The node index in the graph specification. This property is always computed regardless of whether or not it is referenced by the user. The **_id** is a unique identifier which is used to convert the SetCoLa constraints to low-level WebCoLa constraints for the underlying constraint layout.

sources The list of nodes that have edges for which the current node is the target (e.g., all parent nodes of the current node).

targets The list of nodes that have edges for which the current node is the source (e.g., all child nodes of the current node).

neighbors The list of nodes that have edges connected to the current node. This property is the union of the **sources** and **targets** properties. The **neighbors** property may also take an optional value as input, which returns all nodes with a graph distance less than or equal to the specified value.

incoming The list of edges in which the current node is the target (e.g., the list of edges connecting the current node to all sources).

outgoing The list of edges in which the current node is the source (e.g., the list of edges connecting the current node to all targets).

edges The list of edges that contain the current node. This property is the union of the **incoming** and **outgoing** edges.

degree The number of **neighbors** for the current node.

depth One more than the max depth of the node's parents. Root nodes (any nodes with no edges for which the node is the target) have a depth of zero. The **depth** property is only computed for graphs that do not contain cycles.

In the original graph specification, the links are defined by a source and target node. However, whether or not these links are directed or undirected is up to the discretion of the user for how they should be treated in the graph layout. For example, the `neighbors` property is more appropriate than `sources` or `targets` for undirected graphs. This list of properties represents common structural properties that are applicable to a variety of layout specifications. For example, the `depth` property is useful for producing hierarchical tree layouts (e.g., Figure 4.2) whereas the `sources`, `targets`, and `neighbors` properties depict the relationship between nodes as dictated by the graph edges (e.g., Figure 4.6, Line 38). There are many other properties that could be useful for graph layouts that are not included here and this list could easily be extended in the future to include other common properties.

In addition to the built-in properties, we include several operators for manipulating the resulting lists of elements: `length`, `reverse`, `contains`, `sort`, and `extract`. The function `length()` returns the length of the list, `reverse()` reverses the order of the list, and `contains(value)` determines if the list contains the identified value. The user may also choose to `sort` the list based on a property of the elements or `extract` the value of each element for a particular property (e.g., Figure 4.6, Line 38). Values can also be extracted from nodes or edges individually using the dot syntax at any point. The user may use standard array access to extract elements from the list (e.g., the first element of the list is `list[0]`).

4.2.3 *SetCoLa Constraints and WebCoLa Implementation*

Users may specify one or more constraints for each constraint definition. These constraints apply to the nodes within each set produced by the set definition. The current implementation of SetCoLa provides seven constraint types: `alignment`, `position`, `order`, `circle`, `cluster`, `hull`, and `padding`. These constraints were selected to produce a range of expressive graph layouts, which we will further discuss through a series of real-world reproductions in Section 4.3. In the following sections, we explain the design, implementation, and utility of each SetCoLa constraint. We show a sample SetCoLa constraint in each section header.

Graph	Figure	Spec	Nodes	Links	Constraint Definitions	SetCoLa Constraints	WebCoLa Constraints	Ratio (WebCoLa/SetCoLa)
Small Tree	4.2	4.2	• 6	• 5	• 2	• 2	• 11	• 6
Syphilis Social Network	4.4	4.5	• 41	• 74	• 5	• 9	• 166	• 18
TLR4 Network (Cerebral)	4.1	4.6	• 91	• 124	• 5	• 8	• 363	• 45
TLR4 Network (InnateDB)	4.8	4.7	• 98	• 153	• 3	• 4	• 392	• 98
DDX58 Network (InnateDB)	4.9	4.7	• 112	• 187	• 3	• 4	• 448	• 112
NOD-like Network (InnateDB)	4.10	4.7	• 97	• 504	• 3	• 4	• 388	• 97
MAPK1 Network (InnateDB)	4.11	4.7	• 240	• 374	• 3	• 4	• 960	• 240
Kruger Food Web	4.12	4.13	• 16	• 30	• 2	• 3	• 39	• 13
Serengeti Food Web	4.14	4.15	• 161	• 592	• 6	• 19	• 934	• 49

Figure 4.3: The number of nodes, links, and constraints in the layout for each example. The columns labeled **Constraint Definitions** and **SetCoLa Constraints** list the number of SetCoLa set definitions or constraints written by the user, respectively. We compare the number of user-authored **SetCoLa Constraints** to the number of **WebCoLa Constraints** generated by the SetCoLa compiler to determine the factor by which the number of constraints increases (**Ratio**).

The SetCoLa compiler converts each SetCoLa constraint into one or more constraints in WebCoLa [41], which computes the final layout. We leverage two of WebCoLa’s constraints for our implementation: *alignment* constraints and *position* constraints. However, some SetCoLa constraints cannot be directly represented in WebCoLa. For these SetCoLa constraints, we approximate their behavior by imputing additional edges or by applying padding to the nodes. This prototype functionality will be further explained in the following sections. In SetCoLa the user may also define guides to control the layout. In WebCoLa, we add a new node to the graph for each guide and generate constraints relative to this node. These temporary nodes are included in WebCoLa’s layout but are hidden in the final visualization. WebCoLa constraints are defined based on the `_id` of the graph node.

Figure 4.3 shows the number of **Constraint Definitions** and **SetCoLa Constraints** written by the user. We compare the number of **SetCoLa Constraints** to the number of **WebCoLa Constraints** generated by the SetCoLa compiler to show the factor by which the number of constraints increases: **Ratio (WebCoLa/SetCoLa)**. This ratio presents a lower bound on the impact of SetCoLa, since some constraints are not directly converted to WebCoLa, but instead introduce new edges or padding to approximate the layout. In other words, the number of WebCoLa constraints presented in this table is smaller than it would be for a fully implemented constraint specification, which produces more conservative ratios.

Alignment Constraints

EXAMPLE: { "constraint": "align", "axis": "x", "orientation": "top" }

Alignment constraints ensure that all nodes in the set share one of their coordinates. The user must specify the **axis** as either *x* or *y* (Figure 4.2a, Line 18) and may optionally identify an alignment **orientation**. The orientation enables different alignments for elements of varying sizes. The orientation defaults to *center* and aligns the center point of each element. When the alignment axis is defined as *x*, the user may specify the orientation as either *top* or *bottom*, which introduces an offset to align the top or bottom of the elements. When the alignment axis is defined as *y*, the user may specify the orientation as either *right* or *left*.

These constraints are defined as follows. Suppose that the user defines the **axis** as *x* and the **orientation** as *top*. Then, for all nodes n_1 and n_2 in set S such that $n_1 \neq n_2$, we produce the constraint $n_1.y - n_1.height/2 = n_2.y - n_2.height/2$. Analogous constraints are produced for the other possible combinations of axis and orientation. Alignment constraints are one of the constraint types natively supported in WebCoLa. The WebCoLa alignment constraint takes the `_id` of all nodes that should be aligned and offsets for each node, which can be used to change the orientation of the alignment (as discussed above).

Position Constraints

EXAMPLE: { "constraint": "position", "position": "left", "of": "guide1", "gap": 5 }

Position constraints ensure that all nodes in the set are positioned relative to a guide or previously named set. The user must specify the relative placement for the nodes as one of *left*, *right*, *above*, or *below* relative to the guide. The user may optionally define the minimum **gap** between the node and guide (Figure 4.5, Line 11 and 12). These constraints are defined as follows. Suppose the user defines the **position** as *left*, the guide as g , and the **gap** as v . For all nodes $n \in S_1$, we define a constraint that $n.x + v < g.x$. Position constraints are natively supported in WebCoLa and are defined by the node `_ids`, **axis**, and desired **gap**. For each node in set S_1 , we produce one position constraint relative to the specified guide. When the position constraint is defined relative to a named set S_2 , we produce one position constraint for each pair of nodes (u, v) where $u \in S_1$ and $v \in S_2$.

Order Constraints

EXAMPLE: { "constraint": "order", "axis": "y", "by": "depth", "reverse": true }

Order constraints enforce a sort order on the set elements. The user must specify the **axis** on which to sort the elements as either x or y and must also define the node property **by** which the element order is determined (Figure 4.2a, Line 23). The user can optionally define an explicit list of values for a custom **order** (Figure 4.6, Line 23); otherwise, the elements are ordered lexicographically by the specified property. The user may also indicate whether to **reverse** the order of the elements (Figure 4.15, Line 52). Similar to the position constraint, the user may also specify the minimum **gap** between the nodes in the sorted order along the specified axis. These constraints are defined as follows. Suppose the user defines the **axis** as x and the property to sort **by** as *depth*. For all nodes n_1 and n_2 in set S such that $n_1 \neq n_2$ then $n_1.x < n_2.x$ if $n_1.depth < n_2.depth$. We optimize the implementation of this order constraint by only producing constraints between adjacent nodes in the sorted order; in other words, for a set S with n nodes we produce $n - 1$ constraints on the node positions.

When applying constraints to elements that are sets rather than to nodes directly, we create temporary boundary nodes and compute constraints relative to these boundaries. Consider a constraint definition that includes s sets. In this case, we define $s - 1$ boundary guides b_1, b_2, \dots, b_{s-1} . We then identify the order of the sets and produce constraints with the internal nodes for the set. For constraint definition C with s sets, let S_1 and S_2 be two adjacent sets such that $S_1 < S_2$ in the sort order. Let b_1 be the boundary between these two sets. We produce constraints such that for all nodes $n \in S_1$ then $n.x < b_1.x$ and for all nodes $m \in S_2$ then $b_1.x < m.x$. Users may optionally specify a **band** property (Figure 4.5, Line 22) that determines a size for each set region to introduce fixed spacing. In this case, we create $s + 1$ boundary guides and generate constraints at the start and end of the ordering.

The **band** property can be useful for constraining the size of the layout along the specified axis, by limiting the freedom of elements at either end of the ordering. This functionality can be paired with position constraints relative to boundary guides to constrain the overall chart area. For example, Figure 4.6 uses **band** to constrain the individual layers (Line 22) along with position constraints relative to a left (Line 9) and right (Line 10) guide.

Circle Constraints

EXAMPLE: { "constraint": "circle", "around": "center", "radius": 75 }

Circle constraints allow the user to specify a ring layout for a set of elements. The user must define the value `around` which to compute the layout. This value can be either a default *center* or a previously named guide. The user may optionally define a `radius` that defines the expected radius for the circle (Figure 4.5, Line 37).

Circle constraints are not currently supported in WebCoLa, making it difficult to support them directly in SetCoLa. To demonstrate the utility of this constraint type for customized layouts we approximate the behavior as follows. We first add a temporary node or identify the guide node that will act as the center of the circle layout. We then add a link between each node in the set and the center node. Finally, we link the set nodes in the circle with additional temporary edges to produce a chain. We compute the expected length for each edge based on the number of nodes in the circle and the `radius` defined by the user. This strategy approximates a circular layout (Figure 4.4b), though future work should explore the incorporation of alternative strategies to applying constraints that generate circle layouts [47].

Cluster Constraints

EXAMPLE: { "constraint": "cluster" }

Cluster constraints encourage a clustering of the nodes into a compact group by reducing the distance between nodes. This constraint does not currently introduce additional parameters; instead, sets that should be clustered are simply defined as such (Figure 4.15, Line 17). Cluster constraints are not currently supported in WebCoLa. In order to produce a clustered appearance, we add temporary edges between all nodes in the set to produce a clique and require the edges to have a length shorter than the size of the nodes, which pulls the nodes together. These temporary edges remain in the layout but are hidden from the user. Future work should explore how best to encode this stylistic requirement in terms of constraints on the individual nodes; one approach may be to constrain the maximum gap between nodes with respect to the anticipated distance across the cluster. In other words, each pair of nodes in the set should have a gap less than or equal to the desired width of the cluster.

Hull Constraints

EXAMPLE: { "constraint": "hull" }

Hull constraints create an enclosing boundary (hull) around the set elements and prevent other nodes from residing within that boundary (Figure 4.6, Line 31). This constraint produces a visual grouping of nodes that is more strict than the cluster constraint. These constraints are defined as follows. We produce a minimally enclosing rectangle B with properties $B.x1$, $B.x2$, $B.y1$, $B.y2$. For all nodes $n \in S$, we define constraints such that $B.x1 < n.x$, $n.x < B.x2$, $B.y1 < n.y$, and $n.y < B.y2$. For all nodes $m \notin S$, we define constraints that $m.x < B.x1 \parallel B.x2 < m.x$ and $m.y < B.y1 \parallel B.y2 < m.y$. We implement hull constraints in WebCoLa using its built-in support for specifying groups, which produces a boundary around the nodes defined by the `_ids` of nodes in the group.

Padding Constraints

EXAMPLE: { "constraint": "padding", "amount": 5 }

Padding constraints enforce a minimum spacing around an element, without constraining the axis on which the padding is added. In other words, for any pair of nodes the shortest distance between the outer edge of the two nodes should be greater than or equal to the specified padding. The user must define the `amount` of padding that should be added to the node (Figure 4.5, Line 31). Our current implementation adds padding to the node geometry which essentially increases the size of the element when WebCoLa's non-overlap behavior is applied. In this implementation padding can only be specified to a given node once because it is defined in terms of the node geometry itself. Therefore, this padding impacts the spacing relative to all other nodes in the layout. Future work should explore how best to define constraints that respect the desired padding only relative to specific set elements.

4.2.4 Discussion: Application and Program Understanding for Multiple Constraints

The seven constraint types introduced in the previous section enable expressive layouts for several real-world examples (see Section 4.3). However, not all combinations of constraints produce desirable or satisfiable layouts. The current implementation of SetCoLa does not

limit the number or type of constraints that can be applied within a constraint definition. For example, the user could produce contradictions by defining constraints that are the reverse of one another (e.g., two order constraints, one with the ordering reversed). Similarly, applying an alignment constraint to both the x and y axes would require all nodes in the set to share the same position despite overall WebCoLa requirements to avoid node overlap. These concerns are common in constraint systems, and are therefore not limited to SetCoLa.

While some combinations of constraints produce contradictory or overconstrained layouts, many combinations can produce highly expressive layouts. For example, the small tree in Figure 4.2 effectively combines node alignment with a total ordering on the sets to produce a simple specification for a tree layout. Position constraints generally allow the user to arrange the layout relative to global elements, whereas order constraints introduce additional sort requirements between nodes within a particular set. Combining multiple (non-contradictory) position and/or order constraints can allow the user to constrain node positions to particular areas of the visualization, and thus produce overall constraints on the size of the chart area (Figure 4.6, Figure 4.7). This approach can also enable users to introduce distinct regions of interest based on the underlying node properties (Figure 4.4b, Figure 4.5).

One of the challenges explored in this dissertation is to “communicate system behavior as actionable information,” which proves particularly essential and difficult when attempting to debug or understand the behavior of constraints (as discussed in Chapter 3). One advantage of the proposed approach is that the high-level nature of SetCoLa’s constraints can facilitate understanding the source of contradictions with respect to domain-specific properties. We are able to address this challenge by “raising the level of abstraction to reflect user expertise.” In other words, constraints reflect domain-specific properties of the graph data rather than abstract relationships between individual nodes based only on the node `_id`. This example further illustrates how we satisfy the need for systems to “support the tasks that matter most to the user.” With the improved high-level encoding provided by SetCoLa, users can now reason about the properties with which they are most familiar rather than attempting to track or reason about changes to individual nodes in the underlying layout.

4.3 Evaluation: Real-World Examples Reproduced in SetCoLa

To demonstrate the conciseness and expressiveness of SetCoLa for creating domain-specific graph layouts, we reproduce several real-world examples that visualize social networks [164], biological systems [6, 25], and ecological networks [7, 124]. We compare our recreated visualizations to the original layouts and discuss the benefits of our technique for creating highly customized graph layouts. For each recreated example, the layout of the nodes is produced entirely in SetCoLa (e.g., no manual tweaking of the node positions). The nodes in each graph are not given initial starting positions; instead we use WebCoLa to first apply a force-directed layout with no constraints to initialize the graph, before computing the final layout using the constraints produced by the SetCoLa compiler. We manually add labels to the final figures to better match the originals. We include the specification for each of our example layouts (Figure 4.5, 4.6, 4.7, 4.13, 4.15), and annotate the specification with the number of sets produced for each constraint definition (**green**), the number of WebCoLa constraints generated for each SetCoLa constraint (**blue**), and the behavior of the SetCoLa constraints that are not directly translated to WebCoLa constraints (**purple**).

4.3.1 Syphilis Social Network

Social networks can be a powerful way to understand interpersonal relationships and are useful for tracking the spread of diseases that result from personal contact [52, 57, 137, 164]. The ability to track and identify at risk individuals can lead to treatment and help manage the spread of the disease. In addition to the links between individuals, structuring the layout by node properties such as the social or ethnographically-identified group may reveal additional details about how the disease is spread [164].

Rothenberg et al. discuss an ethnographic approach to identify the “core” groups in a social network to better understand the transmission of syphilis amongst sexual partners [164]. Rothenberg et al. found that there were three primary groups involved in the sexual network under study: young affluent white men, younger white women, and young African-American

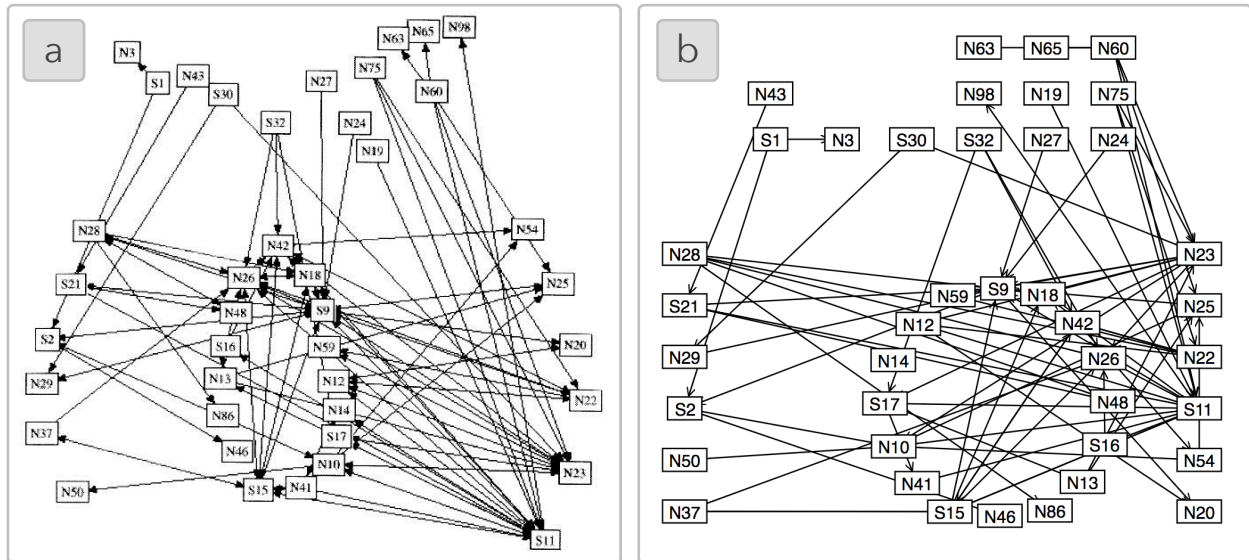


Figure 4.4: The layout for the syphilis social network from (a) Rothenberg et al. [164]. (b) We recreated and improved the layout in SetCoLa by introducing extra padding, alignment, and circle constraints to emphasize the number of interactions between different groups. The nodes are split into three groups, from left to right: young affluent white men, younger white women, and young African-American men. Individuals not included in these “core” groups are positioned at the top. Individuals diagnosed with syphilis during the outbreak are labeled with an “S” on the node.

men, which are visualized from left to right in Figure 4.4a. The authors note that several outsiders to these “core” groups (visualized as the top cluster of Figure 4.4a) played a significant role in the network: *“Visualization of these groups and all their sex partners uncovered the importance of several people not specifically identified with these groups”* [164].

We reproduced this visualization with SetCoLa (Figure 4.4b, 4.5) and included a number of additional constraints on the layout apart from the separation constraints that are visible in the original image. In particular, we included a circle constraint on the group of younger white women to more strongly enforce the result shown in the original figure and applied an alignment constraint on the two groups of young men, as well as some additional padding.

The simplest recreation of this layout uses three constraint definitions and three SetCoLa constraints to produce 123 WebCoLa constraints. Our modified layout (Figure 4.4b, 4.5) includes five constraint definitions and nine SetCoLa constraints, to generate 166 WebCoLa constraints (Figure 4.3). With a small number of user-defined constraints, we can update the layout to more effectively communicate the groupings. The alignment and circle constraints


```

1  "guides": [
2    {"name": "top", "y": 50},
3    {"name": "boundary", "y": 100},
4    {"name": "bottom", "y": 400}
5  ],
6  "constraintDefinitions": [
7    {
8      "name": "ethnographic groups",
9      "sets": {"partition": "group", "exclude": ["other"]},
10     "forEach": [
11       {"constraint": "position", "position": "below", "of": "boundary", "gap": 75},
12       {"constraint": "position", "position": "above", "of": "bottom", "gap": 30 }
13     ]
14   },
15   {
16     "sets": [
17       {"expr": "node.group === 'young white men'"},
18       {"expr": "node.group === 'younger white women' || node.group === 'other'"},
19       {"expr": "node.group === 'young african american men'"}
20     ],
21     "forEach": [
22       {"constraint": "order", "axis": "x", "by": "_exprIndex", "band": 300, "gap": 30}
23     ]
24   },
25   {
26     "name": "other individuals",
27     "sets": {"partition": "group", "include": ["other"]},
28     "forEach": [
29       {"constraint": "position", "position": "below", "of": "top", "gap": 5},
30       {"constraint": "position", "position": "above", "of": "boundary", "gap": 5},
31       {"constraint": "padding", "amount": 8}
32     ]
33   },
34   {
35     "name": "women",
36     "sets": {"partition": "group", "include": ["younger white women"]},
37     "forEach": [{"constraint": "circle", "around": "center", "radius": 75}]
38   },
39   {
40     "name": "men",
41     "sets": {"partition": "group", "exclude": ["younger white women", "other"]},
42     "forEach": [
43       {"constraint": "align", "axis": "y"},
44       {"constraint": "padding", "amount": 10}
45     ]
46   }
47 ]

```

Figure 4.5: The SetCoLa specification for the syphilis social network shown in Figure 4.4. The code is annotated with the number of sets produced (green), the number of WebCoLa constraints generated (blue), and the behavior of SetCoLa constraints not converted to WebCoLa (purple).

emphasize the group relationships by introducing shared visual properties amongst the nodes. The more grid-like layout also facilitates scanning of the nodes to read details included in the node labels, such as whether or not the individual was diagnosed with syphilis (denoted by an “S” in the label). The WebCoLa constraint solver includes a procedure to reduce the length of the edges, which encourages the circle of women to shift towards the group of African-American men (on the right), further demonstrating the relatively larger number of interactions between the two groups, as described in the original paper [164].

```

1  "guides": [
2    {"name": "left_guide", "x": 250},
3    {"name": "right_guide", "x": 750}
4  ],
5  "constraintDefinitions": [
6    {
7      "name": "boundary", 1 set that contains all nodes
8      "forEach": [
9        {"constraint": "position", "position": "right", "of": "left_guide", "gap": 10}, 91 constraints
10       {"constraint": "position", "position": "left", "of": "right_guide", "gap": 10} 91 constraints
11     ]
12   },
13   {
14     "sets": {"partition": "type", "exclude": ["unknown", "downstream genes"]}, 4 sets
15     "forEach": [{"constraint": "padding", "amount": 15}] padding added to 47 nodes
16   },
17   {
18     "sets": [{"partition": "type", "exclude": ["unknown"]}], 1 set
19     "forEach": [
20       {
21         "constraint": "order", "axis": "y", "by": "type", 180 constraints
22         "band": 100, "gap": 30,
23         "order": ["extracellular", ..., "downstream genes"]
24       }
25     ]
26   },
27   {
28     "from": {"expr": "node.type === 'downstream genes'"}, 12 sets
29     "sets": {"partition": "group"},
30     "forEach": [
31       {"constraint": "hull", "style": "visible"}, 12 WebCoLa groups created
32       {"constraint": "cluster"}, 145 new edges added
33       {"constraint": "padding", "amount": 5} padding added to 43 nodes
34     ]
35   },
36   {
37     "from": {"expr": "node.type === 'unknown'"}, 1 set
38     "sets": {"collect": ["node._id", "node.neighbors.extract('_id')"]},
39     "forEach": [{"constraint": "align", "axis": "x"}] 1 constraint
40   }
41 ]

```

Figure 4.6: The SetCoLa specification for the TLR4 biological system shown in Figure 4.1. The code is annotated with the number of sets produced (**green**), the number of WebCoLa constraints generated (**blue**), and the behavior of SetCoLa constraints not converted to WebCoLa (**purple**).

4.3.2 TLR4 Biological Network

Biological networks are a common domain requiring customized visualizations that represent the cellular structure of the nodes in addition to the links contained in the network. Cerebral [6] is a visualization tool designed to show variations in biological networks across experimental conditions. While such layouts are commonly produced by hand, Barsky et al. show that Cerebral can automatically and efficiently arrange the nodes by the location of the biomolecule within a cell (Figure 4.1a). The immune response outcomes are positioned at the bottom of the figure and grouped by biological function. Our reproduction of this graph in SetCoLa is shown in Figure 4.1b, with the specification shown in Figure 4.6.

```

1  "guides": [
2    {"name": "left_guide", "x": 250},
3    {"name": "right_guide", "x": 475}
4  ],
5  "constraintDefinitions": [
6    {
7      "name": "boundary",
8      "forEach": [
9        {"constraint": "position", "position": "right", "of": "left_guide", "gap": 10},
10       {"constraint": "position", "position": "left", "of": "right_guide", "gap": 10}
11      ]
12    },
13    {
14      "name": "cellular location",
15      "sets": {"partition": "Localization"},
16      "forEach": [{"constraint": "padding", "amount": 16}]
17    },
18    {
19      "sets": ["cellular location"],
20      "forEach": [
21        {
28          "constraint": "order", "axis": "y", "by": "Localization",
29          "band": 125, "gap": 30,
30          "order": ["Extracellular", "Cell surface", "Plasma membrane", "Cytoplasm", "Nucleus", "Unknown"]
31        }
32      ]
33    }
34 ]

```

Figure 4.7: The SetCoLa specification for biological networks that can be reapplied across four graphs: TLR4 interaction network (Figure 4.8), the DDX58 interaction network (Figure 4.9), the NOD-like signaling pathway (Figure 4.10), and the MAPK1 interaction network (Figure 4.11). This layout is a simplification of the SetCoLa specification for a TLR4 graph layout shown in Figure 4.6.

Our recreated layout includes five constraint definitions with eight SetCoLa constraints, which generates 363 WebCoLa constraints (Figure 4.3). Similar to Cerebral, this SetCoLa specification could easily be reused across different graphs in the domain since the specification itself does not refer to the individual nodes but to the high-level properties expressed by the layout. We demonstrate such reapplication across several biological networks from InnateDB [25]. InnateDB is a public database containing a large quantity of biological information and is integrated with Cerebral to enable visualization of properties such as protein interactions or signaling pathways. We selected four biological networks from InnateDB to reproduce using SetCoLa: the TLR4 biological system (Figure 4.8)—which shows similar data to the previous example (Figure 4.1)—the DDX58 biological system (Figure 4.9), the NOD-like signaling pathway (Figure 4.10), and the MAPK1 biological system (Figure 4.11). For each example graph, we apply the same SetCoLa specification (Figure 4.7). The biological systems from InnateDB do not include the grouped immune response outcomes, so we use this simplified SetCoLa specification based on Figure 4.6 for the more general graphs.

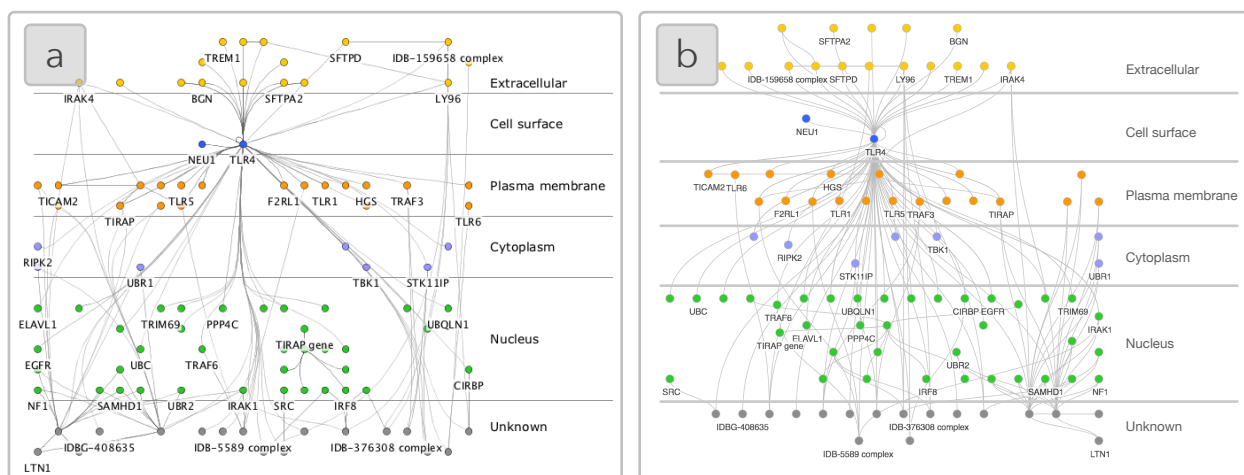


Figure 4.8: The layout for the TLR4 biological system produced using (a) the Cerebral visualization from InnateDB [96] as compared to (b) SetCoLa.

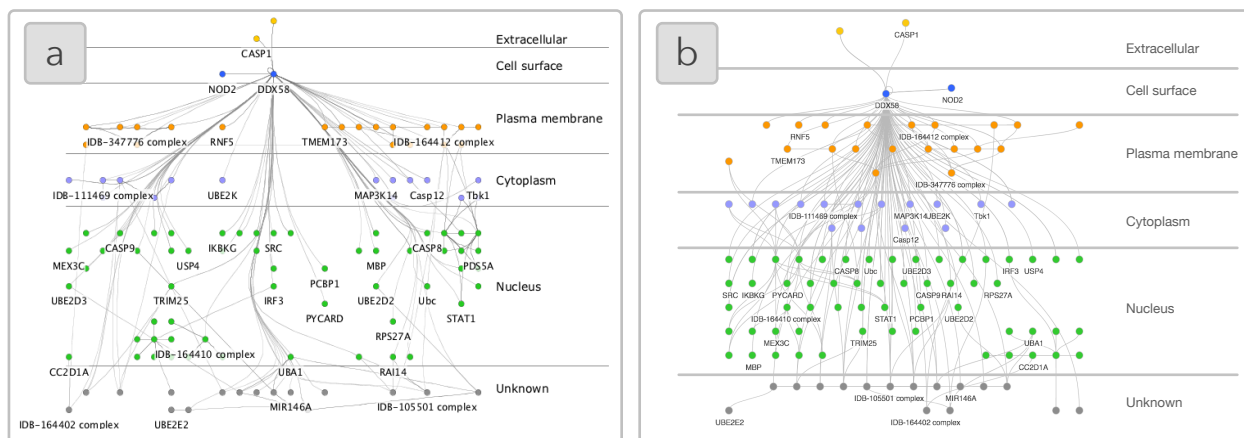


Figure 4.9: The layout for the DDX58 biological system produced using (a) the Cerebral visualization from InnateDB [94] as compared to (b) SetCoLa.

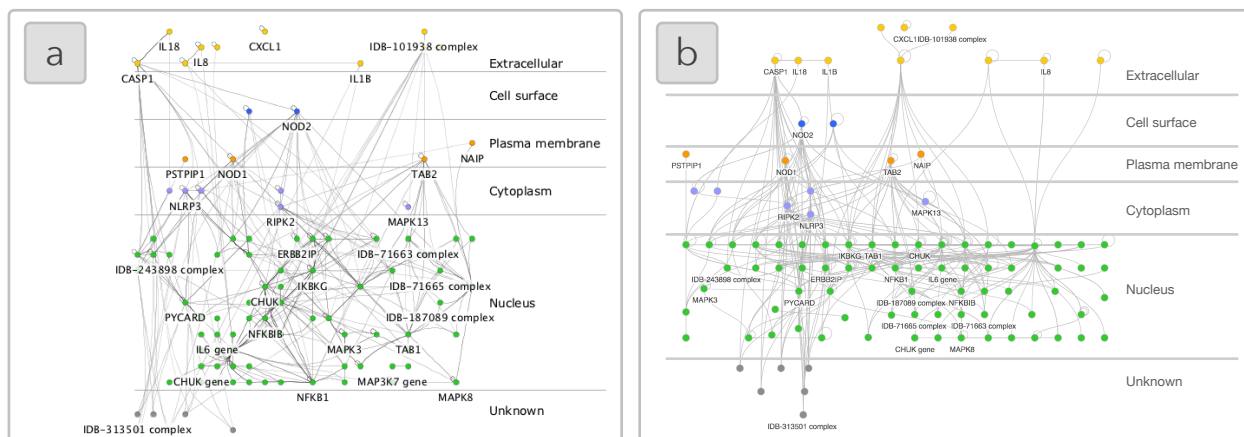


Figure 4.10: The layout for the NOD-like signaling pathway produced using (a) the Cerebral visualization from InnateDB [97] as compared to (b) SetCoLa.

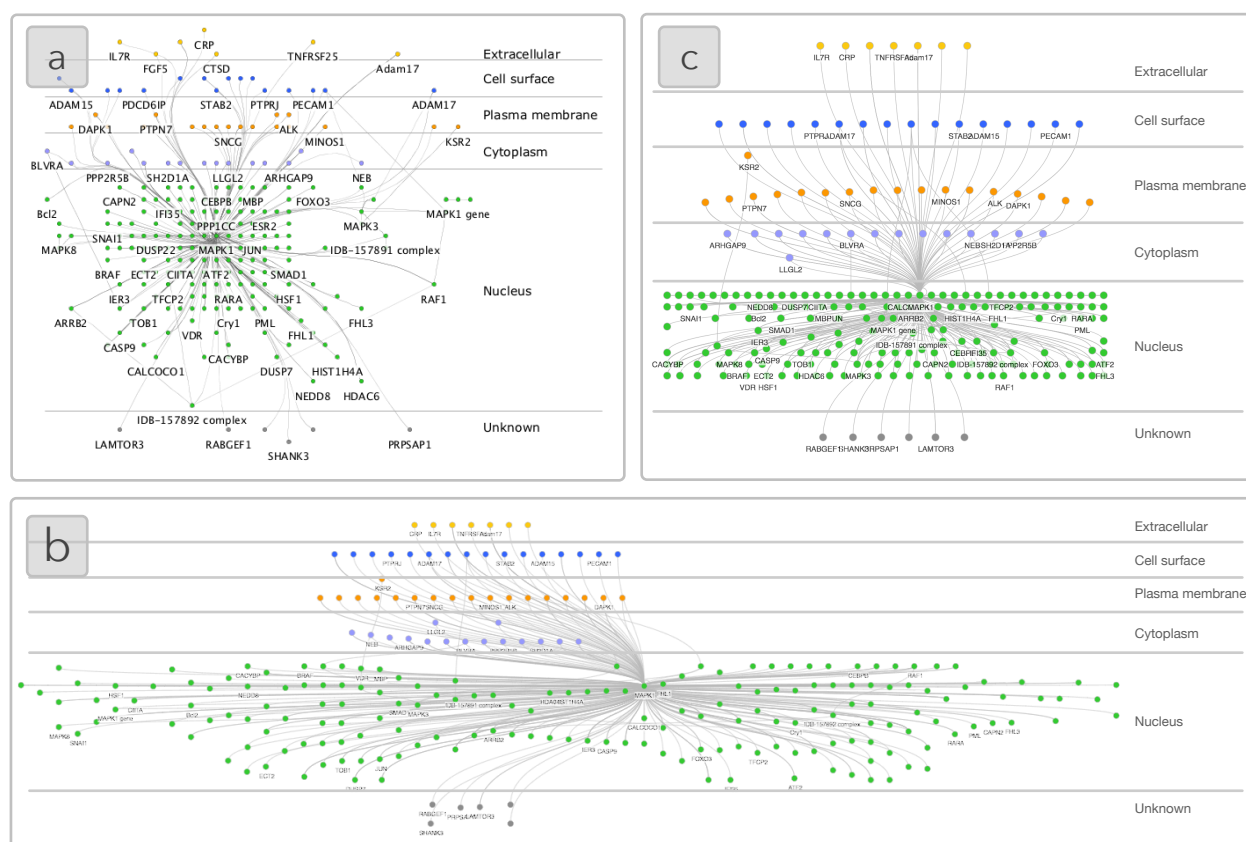


Figure 4.11: The layout for the MAPK1 biological system produced using (a) the Cerebral visualization from InnateDB [95] as compared to (b) SetCoLa. The node padding and constraints on the size of each layer produces an undesirable layout for this larger graph as compared to the smaller alternatives (Figures 4.8, 4.9, 4.10). (c) By adding an additional constraint to reduce the spacing between nodes in the “Nucleus” layer, we can achieve a more desirable layout for this graph.

While the SetCoLa specification works well for the TLR4 network, DDX58 network, and NOD-like signaling pathway, it produces an undesirable result for the MAPK1 network (Figure 4.11b). This layout appears more flattened because it has over twice the number of nodes as the other networks (e.g., 240 nodes as compared to about 100 in the smaller networks). In this case, the constants used for spacing are not ideal for the larger network. Future work should explore better techniques for applying spacing relative to graph properties rather than constant values. An improved version of the MAPK1 network (with reduced spacing on the nodes in the “Nucleus” layer) is shown in Figure 4.11c. One key difference between the SetCoLa and Cerebral layouts is the rendering style for the links. Cerebral uses a bundled routing style, which could be added to SetCoLa in the future to achieve this effect.

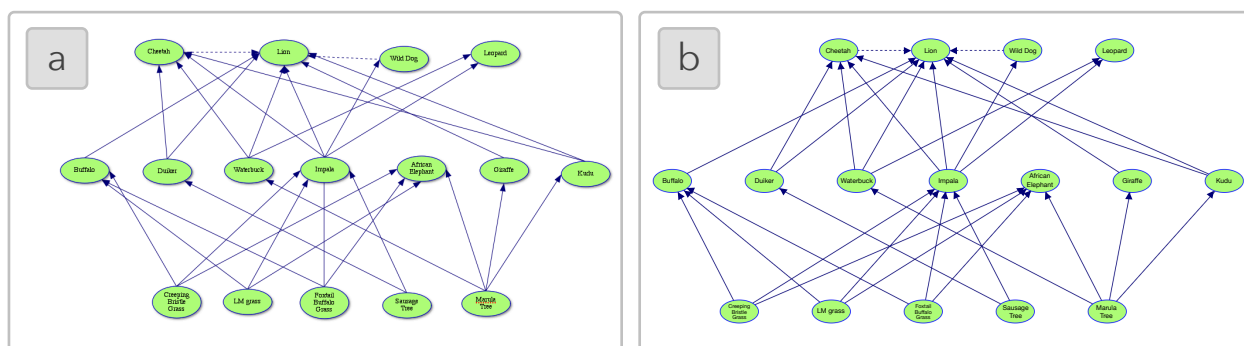


Figure 4.12: (a) A subset of the food web for Kruger National Park arranged by trophic level (i.e., carnivore, herbivore, and plant), as seen on the website [124] and (b) recreated using SetCoLa.

```

1 "constraintDefinitions": [
2   {
3     "name": "tropic_level",
4     "sets": {"partition": "type"},
5     "forEach": [
6       { "constraint": "order", "axis": "x", "by": "order", "gap": 100 },
7       { "constraint": "align", "axis": "x" }
8     ]
9   },
10  {
11    "sets": ["tropic_level"],
12    "forEach": [{
13      "constraint": "order", "axis": "y", "by": "type",
14      "order": ["carnivore", "herbivore", "plant"], "gap": 100,
15    }]
16  }
17 ]

```

Figure 4.13: The SetCoLa specification for the Kruger National Park food web shown in Figure 4.12. The code is annotated with the number of sets produced for each set definition (green) and the number of WebCoLa constraints generated for each SetCoLa constraint (blue).

4.3.3 Kruger National Park and Serengeti National Park Food Webs

Food webs visualize complex producer-consumer relationships in ecological systems based on relevant domain-specific properties for the ecosystem. Despite the challenges in creating an informative visualization [103], food webs are a common presentation strategy for this information [7, 18, 32, 82, 106, 107, 124, 126, 166, 210]. Small or simplified food webs may be drawn by hand, but many real world ecosystems can have hundreds of interconnected organisms. In such cases, a customized layout may be useful for reasoning about the structure of the ecological system by leveraging properties specific to the type of ecosystem, such as the hierarchy of organisms in the food chain or the depth at which oceanic organisms reside.

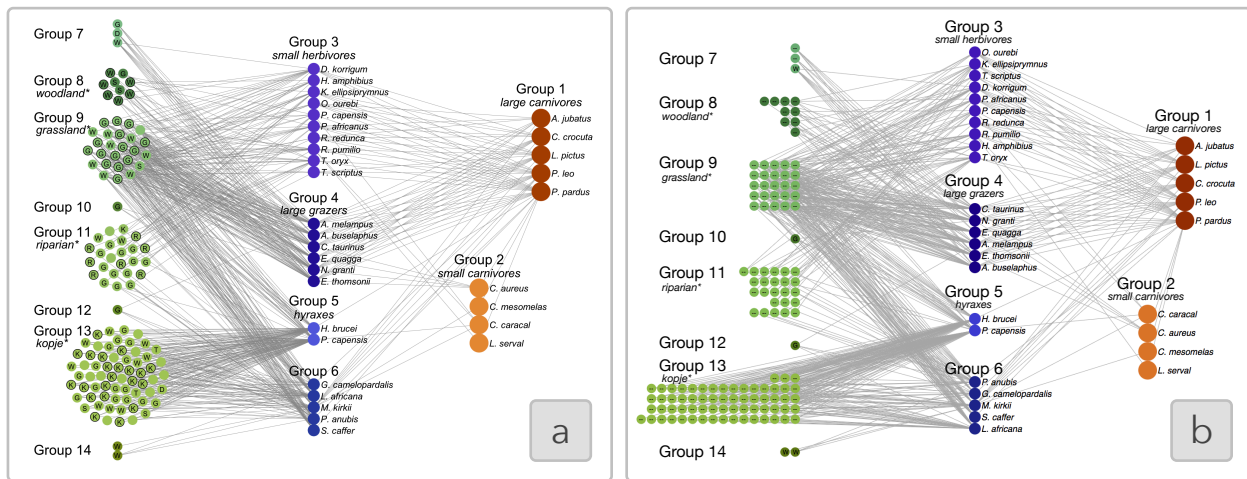


Figure 4.14: The layout for the Serengeti food web from (a) Baskerville et al. [7] as compared to (b) the layout recreated with SetCoLa. Nodes are layered by trophic level (e.g., plant, herbivore, carnivore) and clustered into groups using a Bayesian analysis method.

Small food webs exhibit several of the properties of larger food webs, such as a node hierarchy arranged by the node's trophic level (e.g., the element's role within the food web). For example, Figure 4.12a visualizes a subset of the species found in Kruger National Park [124]. We can easily recreate the layout (Figure 4.12b) with a small number of constraints on the nodes; for the SetCoLa specification (Figure 4.13), we include two constraint definitions with three constraints, which generates 39 WebCoLa constraints (Figure 4.3). In particular, we constrain each trophic level to be aligned and enforce an ordering of the layers that respects the food web hierarchy. We also include a constraint to order each layer by a predefined order property on the nodes to exactly match the original visualization; this particular constraint would therefore be unnecessary for a more generic layout. This SetCoLa specification could easily be applied to other small food webs to produce a similar layout. However, as the food web gets more complex with more nodes associated with each trophic level, it may become necessary to relax the alignment constraints or introduce additional clustering to highlight other structures or domain-specific properties within the layout.

The Serengeti food web from Baskerville et al. [7, 8] is an example of a larger ecological network, which depicts the relationships among 161 plants, herbivores, and carnivores with 592 links between entities. Baskerville et al. employ a Bayesian analysis method to produce

```

1  "guides": [
2    {"name": "top_guide", "y": 0},
3    {"name": "bottom_guide", "y": 100},
4    {"name": "carnivore_guide", "x": 450},
5    {"name": "herbivore_guide", "x": 300},
6    {"name": "plant_guide", "x": 125}
7  ], "constraintDefinitions": [
8    {
9      "name": "plants", 8 sets
10     "sets": {"partition": "group1", "include": [7,8,9,10,11,12,13,14]},
11     "forEach": [
12       {"constraint": "position", "position": "right", "of": "plantG", "gap": 185}, 129 constraints
13       {"constraint": "position", "position": "left", "of": "herbivoreG", "gap": 200}, 129 constraints
14       {"constraint": "position", "position": "below", "of": "topG", "gap": 0}, 129 constraints
15       {"constraint": "position", "position": "above", "of": "bottomG", "gap": 0}, 129 constraints
16       {"constraint": "padding", "amount": 1}, padding added to 129 nodes
17       {"constraint": "cluster"} 2632 new edges added
18     ]
19   },
20   {
21     "sets": ["plants"], 1 set
22     "forEach": [{"constraint": "order", "axis": "y", "by": "group1", "gap": 25}] 253 constraints
23   },
24   {
25     "name": "herbivores", 4 sets
26     "sets": {"partition": "group1", "include": [3,4,5,6]},
27     "forEach": [
28       {"constraint": "align", "axis": "y"}, 4 constraints
29       {"constraint": "position", "position": "right", "of": "herbivoreG", "gap": 185}, 23 constraints
30       {"constraint": "position", "position": "left", "of": "carnivoreG", "gap": 185}, 23 constraints
31       {"constraint": "position", "position": "below", "of": "topG", "gap": 250}, 23 constraints
32       {"constraint": "position", "position": "above", "of": "bottomG", "gap": 50}, 23 constraints
33       {"constraint": "hull"} 4 WebCoLa groups created
34     ]
35   },
36   {
37     "sets": ["herbivores"], 1 set
38     "forEach": [{"constraint": "order", "axis": "y", "by": "group1", "gap": 75}] 31 constraints
39   },
40   {
41     "name": "carnivores", 2 sets
42     "sets": {"partition": "group1", "include": [1,2]},
43     "forEach": [
44       {"constraint": "align", "axis": "y"}, 2 constraints
45       {"constraint": "position", "position": "right", "of": "carnivoreG", "gap": 185}, 9 constraints
46       {"constraint": "position", "position": "below", "of": "topG", "gap": 450} 9 constraints
47     ]
48   },
49   {
50     "sets": ["carnivores"], 1 set
51     "forEach": [
52       {"constraint": "order", "axis": "x", "by": "group1", "reverse": true, "gap": 40}, 9 constraints
53       {"constraint": "order", "axis": "y", "by": "group1", "gap": 40} 9 constraints
54     ]
55   }
56 ]

```

Figure 4.15: The SetCoLa specification for the Serengeti food web shown in Figure 4.14. The code is annotated with the number of sets produced (green), the number of WebCoLa constraints generated (blue), and the behavior of SetCoLa constraints not converted to WebCoLa (purple).

related clusters in each trophic level. Baskerville et al. visualize the results with a customized layout showing both the trophic hierarchy and group clustering (Figure 4.14a). The Bayesian analysis approach and customized visualization highlights relationships between the plant habitats and underlying network structure that may be hard to identify from the data alone.

We reproduce this layout in SetCoLa (Figure 4.14b). For this specification (Figure 4.15), we author six constraint definitions to create sets for each group in the layout and sets representing each trophic level. For the carnivores and herbivores, we constrain the position of the nodes within the visualization region, introduce alignments on the nodes, and manage the order in which the sets are displayed. For the plant sets, we apply cluster constraints to enforce a tighter grouping of the nodes. For this specification, we define a total of 19 SetCoLa constraints, which generate 934 WebCoLa constraints (Figure 4.3). One noticeable difference in the SetCoLa layout is that the plant nodes (groups 7–14) form grid-like rather than “organic” clusters. This behavior results from the current implementation of the cluster constraint, which approximates the layout by adding additional edges. The node positions are also impacted by various position constraints, the edges to nodes outside each group, and WebCoLa’s non-overlap constraint. Future work should explore new approaches that can support a similarly organic layout of nodes in a cluster.

Baskerville et al. [7] note that “*We have not included invertebrates (insects and parasitic helminths) or birds*” in their published food web, though they “*hypothesize that the general conclusions will be largely robust to the addition of more species.*” One advantage of SetCoLa is that the layout is independent of the individual nodes. Therefore, the authors could reuse this SetCoLa layout to visualize future iterations of the Serengeti food web or explore similar structures across different ecological communities.

4.4 Limitations and Future Work

We have presented SetCoLa: a domain-specific language for specifying high-level constraints for customized graph layout. SetCoLa enables concise specification of layouts by applying constraints to node sets rather than individual nodes. These custom layouts can be reapplied to different graphs that share domain-specific properties. We implemented SetCoLa using the WebCoLa library [41] and demonstrated the expressiveness of SetCoLa on real-world examples from ecological networks, biological systems, and social networks. SetCoLa specifications reduce the number of user-authored constraints by one to two orders of magnitude, while

enabling flexible and reusable domain-specific layouts. There are a number of useful areas for future work, including optimizations for the current constraint generation procedure, the development of useful debugging tools to facilitate the user’s understanding of unsatisfiable constraints, and the evolution of constraint solvers more closely integrated with SetCoLa.

4.4.1 Prototyping and Constraint Generation

SetCoLa allows users to define constraints that apply to groups of nodes rather than applying constraints to individual nodes one at a time. By deferring this specification complexity to the underlying constraint solver, the user can more easily prototype the layout and make changes that have a large overall impact with a small number of written constraints. However, the current SetCoLa compiler is not optimized to reduce the number of constraints produced. In particular, the procedure for generating order constraints adds potentially superfluous inter-node constraints. For the small tree example (Figure 4.2c), the SetCoLa compiler produces 11 WebCoLa constraints. The SetCoLa compiler creates two constraints: $b.y < boundary1.y$ (Line 31) and $c.y < boundary1.y$ (Line 33), in addition to a constraint that $b.y == c.y$ (Line 21-24), thus making one of the first two constraints superfluous. Future work should explore whether these redundant constraints have a significant performance impact and investigate optimizations to reduce the generation of unnecessary constraints.

4.4.2 Debugging and Unsatisfiable Constraints in SetCoLa

SetCoLa specifications may include sets that are *not* disjoint, which can produce unsatisfiable constraints. The user may also specify unsatisfiable constraints indirectly through combinations of set definitions and constraint applications. Finally, some specifications may be underconstrained and thus produce layouts that do not meet the user’s expectations. Concerns surrounding debugging and unsatisfiable constraints are not exclusive to SetCoLa, and can also arise in WebCoLa and other constraint-based systems (see Chapter 3).

For the constraints described in this chapter, it is possible to determine if conflicts arise at program runtime and highlight such conflicts. One advantage of the SetCoLa abstraction

is that the original user-authored constraints are defined on the high-level properties of the nodes, which makes it easier to understand why conflicts occur. In order to debug the constraints, the user may first inspect the sets produced by SetCoLa to check for inconsistencies. By identifying nodes that exist in multiple sets, users can more easily understand the source of potential conflicts. Each WebCoLa constraint generated by the SetCoLa compiler is annotated with the SetCoLa constraint from which it was generated; these annotations help “communicate system behavior as actionable information” by allowing users to map more easily between the output constraints and the SetCoLa constraints from which they were generated. While these properties may help with the debugging process, future work should explore additional strategies for debugging the graph layout and underlying constraints.

4.4.3 Limitations of SetCoLa’s Expressiveness

The current SetCoLa implementation requires the graph to be fully formed at input, including all properties (beyond the ones computed in Section 4.2.2). All the edges and nodes in this graph are treated with equal weight in terms of the constraints, thus limiting the user’s ability to introduce preferences regarding the importance of the nodes or links. Furthermore, there are cases in which the user may want to break links, duplicate parts of the graph, or otherwise modify the underlying structure based on properties of interest. The current SetCoLa implementation does not support operations to modify the importance or structure of the input graph, though this functionality would be an interesting area for future work.

4.4.4 Limitations Arising from the Constraint Solver

Our implementation with WebCoLa allows us to demonstrate the utility of SetCoLa for creating reusable, customized domain-specific layouts. However, our current implementation was limited in part by what WebCoLa currently supports. For example, we were unable to directly express SetCoLa’s circle constraints in WebCoLa. However, circle constraints have been identified in the WebCoLa wiki as an area for future work, and once they are supported in the underlying constraint solver, it should be straightforward to use this improved support.

Our work contributes new strategies for the specification of graph layout constraints, but does not aim to create its own constraint solver. WebCoLa is a useful library on which to build and demonstrate our approach, but future work might explore how new or existing constraint solvers might co-evolve alongside this high-level language for constraint specification.

We also encountered some behavioral mismatches between the WebCoLa implementation and our expectations for the graph layout. For example, WebCoLa utilizes a default link length which attempts to optimize node positions to produce links as close to the desired link length as possible. While this technique can be useful for highlighting the underlying structure of the graph, it has a significant effect on the layout that is produced by WebCoLa that may vary from what is specified in SetCoLa. This behavior can be beneficial for some layouts. As noted in Section 4.3.1, in the syphilis social network (Figure 4.4b), the circle is drawn slightly off center between the groups since more links exist between the women and the African-American men than between the women and the white men, which emphasizes the strength of these connections. The underlying constraint solver can thus significantly impact the resulting layout by implicitly encoding additional preferences, such as the preferred link length. In future work, it may be useful to support additional parameters expressing global preferences for the layout, which would then be passed on to the underlying solver. Another direction would be to accommodate multiple solvers, which might encode different preferences of these kinds, and to select among them, either automatically or as specified by the user.

Another useful WebCoLa behavior is that a global non-overlap constraint may be applied to the nodes as part of the layout procedure. This constraint prevents nodes from overlapping even at intermediate stages of the layout and may thus cause the layout to become stuck in a local optimum that still includes unsatisfied constraints. Furthermore, our use of dummy guide nodes with a fixed position may further complicate issues with local maxima. Future work might explore additional procedures for iteratively adding constraints to the graph layout, building up the final result incrementally. This technique would help to reduce the burden on the layout to resolve all constraints at once and allow the user to incrementally improve the layout through the addition of new constraints that restart the underlying solver.

4.5 Summary of Contributions

By using an appropriate graph layout, node-link diagrams can effectively convey properties of the underlying graph structure, such as the node hierarchy or network connectedness. Such visualizations are common across many domains, including social networks [52, 57, 137, 164, 179], biological systems [6, 16, 61, 120, 127, 170, 181], and ecological networks [7, 18, 32, 82, 103, 124, 126, 166, 210]. These graph layouts utilize domain-specific properties to emphasize relevant patterns in the data. In a biological pathway for example, nodes can be layered by their subcellular location to visualize the interconnectedness of the network relative to the cellular structure. The design of effective graph layouts is highly dependent on the domain-expertise of the visualization designer, but effective programming systems often remain out of reach. In this chapter, we show how raising the level of abstraction for layout authoring can support users in encoding their unique domain expertise with reduced programmatic effort as compared to other approaches, and can further facilitate program understanding.

To this end, we contribute SetCoLa: a high-level language for specifying graph layout constraints. SetCoLa aims to facilitate the process of authoring customized layouts by leveraging the domain expertise of the domain expert. Using SetCoLa, the domain expert can focus on mapping domain-specific properties directly to the layout parameters of interest, while deferring the low-level implementation details to the SetCoLa compiler and underlying constraint engine. With this approach, SetCoLa is able to reduce the number of user-authored constraints by one to two orders of magnitude. Furthermore, SetCoLa enables reuse of customized layouts across graphs in the same domain because the constraints are generalized to apply to properties of the network rather than the implementation specifics for the individual nodes (such as the internal node ID). This mapping further improves how users understand or debug the behavior of the constraint layout; constraints are now clearly grounded by the user’s domain expertise to better support reasoning about the encoded relationships.

This work was done in collaboration with Alan Borning and Jeffrey Heer, and was originally published and presented at EuroVis 2018 [83].

Chapter 5

PROGRAM UNDERSTANDING IN VEGA: A DECLARATIVE VISUALIZATION GRAMMAR

Declarative languages introduce a trade-off between flexibility and comprehensibility. By raising the level of abstraction, declarative languages can enable users to encode their unique expertise while focusing on the tasks that matter most to them. For example, Chapter 4 explored how SetCoLa can facilitate the design of customized graph layouts based on domain-specific properties of the data and support reuse of the layouts across graphs in the same domain. However, one challenge that remained in SetCoLa—as well as the other constraint systems discussed in Chapter 3—was interpreting the behavior of the underlying constraints. This challenge further arises across declarative programming languages. Separating the user specification from the system execution obfuscates the underlying program behavior and can inhibit the developer’s ability to evaluate and debug the program output.

To explore the types of challenges that arise for program understanding in declarative programming languages, this chapter examines the design of Vega: a declarative visualization grammar [177, 178]. Similar to SetCoLa [83], Vega aims to raise the level of abstraction to allow users to focus on visualization authoring, particularly for interactive visualization designs. Interaction plays an essential role in visualization design, since interaction techniques such as filtering, brushing, and dynamic queries can facilitate data exploration and understanding [80, 153]. However, implementing such interactions has traditionally required event callbacks, which necessitate manually tracking interleaved state changes [145]. In response, Vega leverages event-driven functional reactive programming [201] to provide declarative primitives for interaction design. This approach models input events as data streams, which in turn drive dynamic variables called *signals*. Signals parameterize the visualization, endow-

ing transforms, scales, and marks with reactivity. When new input events fire, corresponding signals are automatically re-evaluated, updates are propagated to the visual encodings, and the visualization is re-rendered. By deferring the low-level control flow to the system, Vega enables rapid iteration of encoding and interaction design. However, identifying how changes to the specification impact the output or how user interactions with the output and data are propagated through the execution is particularly difficult for these time-varying behaviors.

In this chapter we first provide some relevant background on the design and terminology associated with Vega [177] and the underlying reactive semantics. Inspired by prior work on program visualization and debugging (see Section 2.3), we contribute a data flow graph visualization of the Vega runtime behavior that aims to illustrate the underlying control flow of the system in terms of these semantics. We then contribute a set of formative interviews with expert Vega users about the utility of the proposed data flow graph visualization. This work aims to better understand the challenges that arise when raising the level of abstraction and new opportunities to better communicate the underlying system behavior as actionable information for potential end-user programmers. Through the formative interviews, we explore a trade-off for program understanding techniques between accurately reflecting the system behavior and providing useful debugging features that reduce the burden on programmers and enables them to focus on their primary development tasks.

5.1 Related Work: Functional Reactive Programming

Event-Driven Functional Reactive Programming (E-FRP) [201], one of many FRP variants [5], is an increasingly popular paradigm for authoring interactive behaviors. E-FRP models low-level input events as continuous streams of data, which can be composed into dynamic variables called *signals*. When a new event fires, the E-FRP runtime propagates the update to the corresponding streams, and dependent signals are updated in two phases. In the first phase, signals are re-evaluated using their dependencies' prior values; these dependencies are then re-evaluated in the second phase [201]. E-FRP has been shown to be suitably expressive for interactive web applications [36, 141] and visualizations [35, 108, 177].

However, debugging support remains weak. Many existing debugging techniques—such as breakpoints and stack traces—no longer apply, as users *declaratively* specify interactions. The E-FRP runtime is entirely responsible for the program execution, the particulars of which will be unfamiliar to end users. In this space, the Elm language [36] began to develop an interactive debugger, inspired by Bret Victor [197]. The Elm debugger allows users to record and replay program states, but developers must manually annotate their code with `watch` and `trace` statements. Tracked states are then simply printed out in a list.

5.2 Background and Terminology for the Vega Visualization Grammar

To explore program understanding techniques for declarative programming languages we focus on the design of Vega [177]: a declarative visualization grammar for interactive visualization design. In this section we provide some relevant background information on the design and behavior of the Vega visualization grammar to inform our discussion of new program understanding techniques for reactive systems. This chapter further introduces relevant terminology that will apply across the remaining chapters in this dissertation.

Vega [177] is a declarative grammar for specifying interactive visualizations. The programmer produces a Vega *specification* in JSON format that describes the data transformations, interactive behavior, and visual appearance for an output visualization. A JavaScript runtime parses the input specification to produce the resulting visualization. To construct a visualization, the programmer must first include the *datasets* of interest. *Datasets* represent collections of data tuples and can be representative of more complex data structures with arbitrary nesting and usage throughout the code. References to particular *data fields* extract a property from each tuple in the underlying datasets to be used as variables throughout the Vega specification. Integrated *data transformation* pipelines provide operations including statistical summarization and spatial layout (e.g., treemaps and cartographic projections).

Closely following the model of Protovis [22] and D3.js [23], the visual appearance of the Vega visualization is specified via *scales*, *axes*, *legends*, and graphical primitives called *marks*. *Scales* are functions that map from data values to visual properties, and can themselves be

visualized as *guides* (e.g., *axes* and *legends*). *Marks* are graphical primitives such as *bars*, plotting *symbols*, and *lines*; marks can be arbitrarily nested and dynamically initialized at runtime, thus introducing complex data flows during program execution. The properties of *marks* (e.g., the position or color) can be parameterized by both *signals* and *data*, with the help of *scales* to produce reasonable mappings from *data fields* to visual properties.

To support interaction design, Vega employs Event-Driven Functional Reactive Programming (E-FRP) [177]. Input events are modeled as streams of data, and an event selector syntax facilitates stream composition. *Signals* are in turn defined as reactive expressions over stream values. For instance, a signal might extract the x and y coordinates from the most recent mouse input event. Signal values defined in pixel space can be passed through inverse scale transforms to map the coordinates back to the data domain. Scale inversions allow interactive behaviors to generalize across distinct coordinate spaces (e.g., small multiples) or coordinate interaction across multiple visualizations (e.g., brushing and linking).

Signals can parameterize the remainder of the Vega specification, thereby endowing data transformations and visual encodings with reactive semantics. Reactive updates (referred to as *pulses*) occur in two steps. When an event occurs, dependent signals are re-evaluated in their specification order. This step allows signal expressions to access the previous values of dependencies listed later in the specification; these dependencies are subsequently updated on the same pulse. Once the signals have updated, the dependent data transformations and visual encodings are recomputed in topological order of the underlying dependency graph.

Signals decouple low-level input events from interaction logic. For example, the same set of named signals can be driven by mouse and touch events. Moreover, signals express the bulk of the interaction logic and participate in visual encodings either as direct parameters or by parameterizing simple if-then-else encoding rules. As a result, signals provide a meaningful entry-point into an interaction specification. In contrast to imperative event handlers, complex static analysis is not required to identify and surface the relevant program state.

5.2.1 *Discussion on the Design of Vega*

The Vega visualization grammar allows end-user programmers to focus on relevant visual encoding decisions for the design of an interactive visualization, while deferring the execution of this design to the underlying Vega runtime. Similar to other declarative programming languages, this approach enables users to employ their personal expertise while focusing on their primary development task. However, the separation between user-authored code and system produced output complicates the program understanding process. Vega provides a unique opportunity to explore complex program understanding topics for a reactive programming domain: interactive visualization design. The semantics of Vega reflect those of similar reactive programming languages, such as React [93] and Elm [36].

One advantage of Vega as an environment in which to explore program understanding is that the core Vega constructs—such as the datasets, data fields, and interactive signals—fully encapsulate the interactive behavior of the code. These constructs provide a clear entry point for new debugging opportunities, which we further explore in Chapters 6 and 7. The time-varying behavior can be particularly difficult to understand and debug, but the provided constructs can facilitate snapshotting and replay of the full program functionality. One of the main challenges is therefore to “communicate system behavior as actionable information” that can help reduce the gap between the user-authored code and system produced output.

5.3 *Visualizing the Vega Runtime Behavior as a Data Flow Graph*

Vega accepts a JSON specification that is parsed into a data flow graph representing the execution pipeline. Data tuples are pushed through the data flow graph to be rendered into the output visualization. Prior to this work, there was no infrastructure for debugging visualizations in Vega. Users could only rely on the JavaScript console to traverse the underlying system internals. However, accessing and navigating the system internals requires existing knowledge of how to locate relevant information, which is often deeply nested in the internal structure. This structure also contains extraneous details that complicate identification of relevant information. The structural disconnect between signals, data, and encodings makes

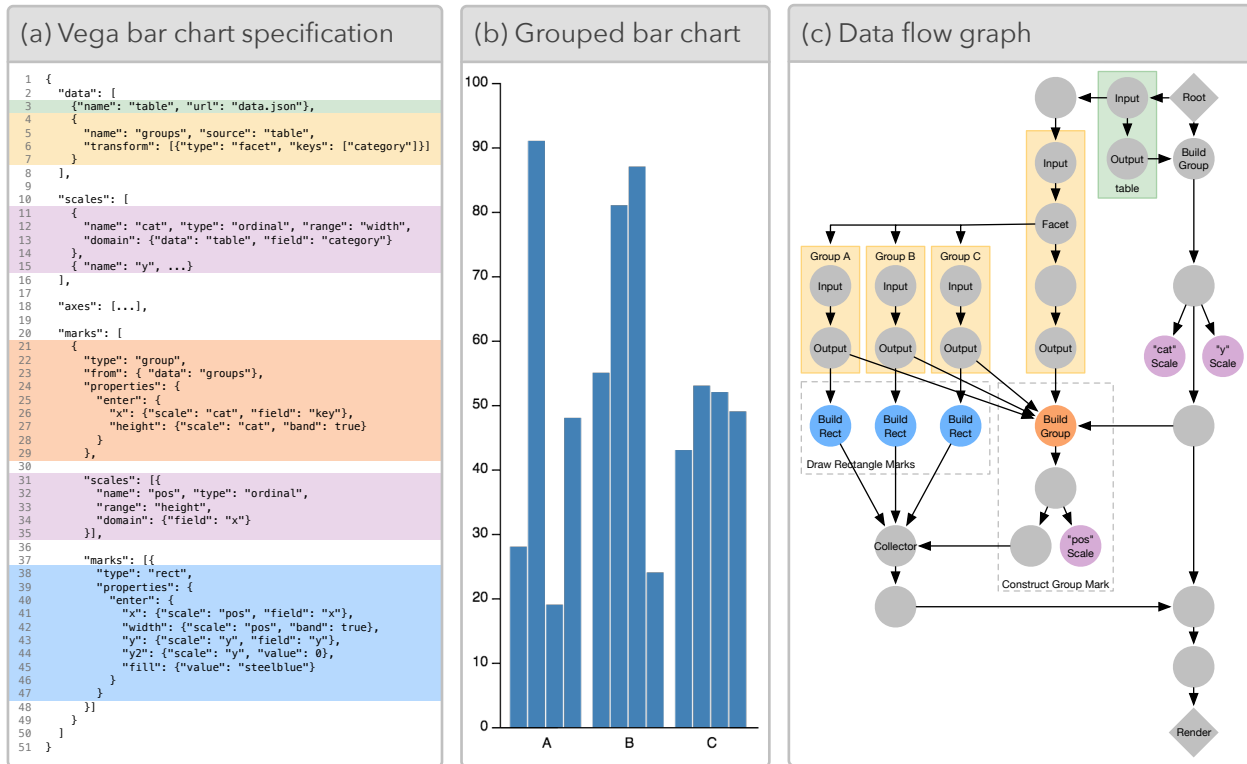


Figure 5.1: The components of a Vega workflow: (a) the Vega specification for a grouped bar chart, (b) the output visualization, and (c) a simplified representation of the underlying data flow graph. The specification highlights the code corresponding to different parts of the data flow graph.

it hard to track changes between components, and is impractical for complex tasks. An example of the debugging process using this approach is available in Appendix B. However, this method is not discoverable or intuitive for novice users. For developers without this expertise, the specification and output visualization are the only sources available for debugging, but neither provide insight into the underlying structure or how the two pieces relate.

Usage Scenario. Consider a scenario in which a user wants to implement a grouped bar chart, but the initial specification produces a blank chart showing only the incomplete axes. Upon inspecting the data flow graph, the user realizes that the bars are being drawn from the root of a hierarchical data source, not from each of the groups produced by the facet transformation. To resolve the error, the user notes that the rectangle marks must inherit from a group mark to unpack the hierarchical structure. This hierarchical structure is clearer in the data flow graph visualization (Figure 5.1c) than in the specification alone (Figure 5.1a).

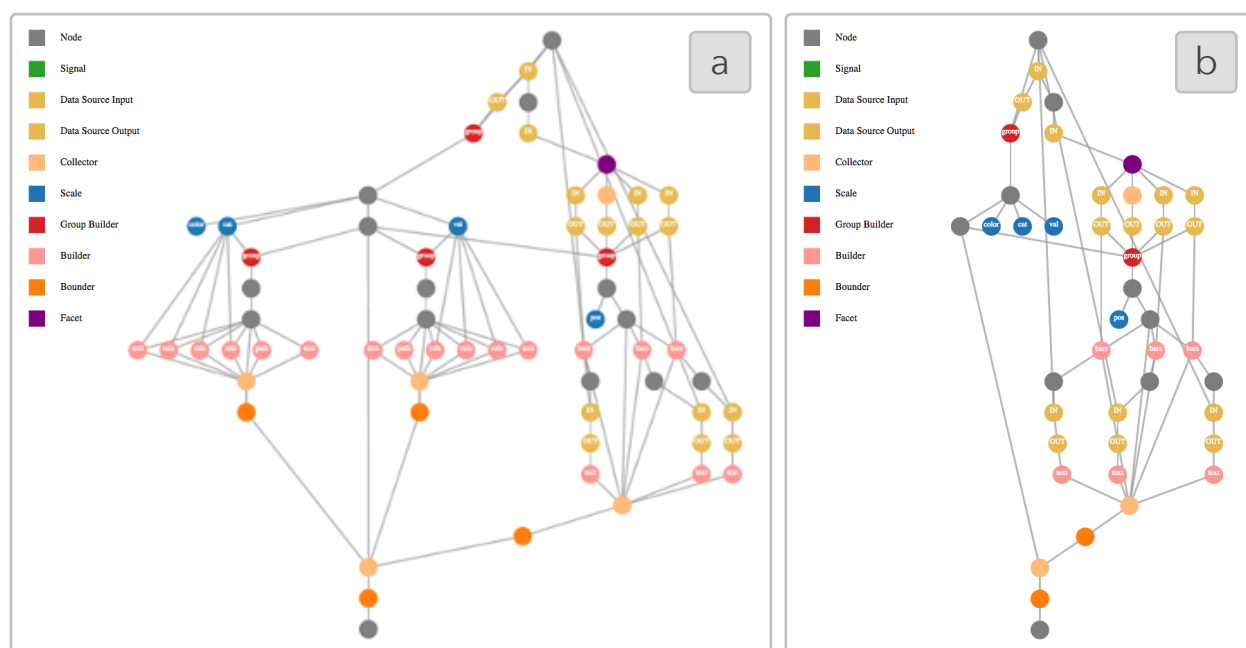


Figure 5.2: (a) The prototype data flow graph visualization for a grouped bar in Vega. (b) The same data flow graph visualization with nodes corresponding to the axes removed. This simplification produces a smaller graph that is more focused on the structure of the relevant visualization marks.

To provide insight into the Vega system structure, we can visualize and annotate the underlying data flow graph (Figure 5.1c). The simplest version of this graph represents the entirety of the execution structure for rendering a Vega specification. However, an accurate portrayal of the data flow graph contains intermediate nodes and edges, such as the “Collector” and unlabeled nodes of Figure 5.1c. These nodes are artifacts of Vega’s low-level implementation and internal system optimizations. For example, the “Build Group” node (shown in orange) is responsible for constructing the group mark container for the rectangle marks. However, the “Build Group” node is not directly connected to the nodes drawing the rectangle marks (shown in blue) despite this hierarchical relationship. Instead, all nodes are routed through a “Collector.” This extended internal relationship makes it hard to infer from the visualization how the rectangle marks are related to the group mark and the faceted data source. While these connections may be intuitive to Vega system developers, they do not have a clear connection to the components with which end-user programmers will be most familiar, such as the specification (Figure 5.1a) and output visualization (Figure 5.1b).

To better understand the potential utility of the data flow graph, we developed a prototype visualization that automatically extracts the data flow structure for any output Vega visualization. An example of the output data flow graph is included in Figure 5.2. There are several challenges that arise when considering this structure. First, the behavior of axes and legends introduces superfluous detail to the data flow structure. Axes and legends are automatically generated in Vega and are constructed based on several components including text marks, tick marks, domain line, gridlines, and title. These components add complexity to the data flow graph, but are rarely the most interesting component for an end user to examine. Figure 5.2a shows the data flow graph with the axis components visible, whereas Figure 5.2b simplifies the representation to remove the components related to each axis; in the simplified version of the data flow graph, users can more easily focus on the relationships produced by the facet transformation (shown in **purple**), as well as the subsequent data pipelines for each of the three facets produced. These representations also include a variety of nodes specific to the underlying system behavior—such as the collectors (shown in **pale orange**) or the other unlabeled nodes (shown in **gray**)—which are likely unfamiliar to the end-user programmer. Finally, these visualizations may include many repeated structures. For example, the facet behavior used to produce a grouped bar chart introduces multiple duplicate pipelines in the data flow graph depending on the number of facets that are produced.

To leverage the informative power of the data flow graph while addressing these complications, we explored a variety of customized representations of the data flow graph structure. Figure 5.1c shows a simplified and stylized version of the data flow graphs from Figure 5.2. In this structure, we introduced additional groupings to highlight the repeated structures in the graph and to label essential components of the data flow structure—such as the nodes responsible for drawing the “rect” marks of the output visualization. For expert Vega developers, these relationships can be extracted from the more general data flow graph, but the annotations are important for ensuring the legibility of this structure for novice Vega users. To illustrate the underlying functionality of Vega, we incorporate a variety of these stylized data flow graph representations into the Reactive Vega paper published at VIS 2015 [177].

5.4 Formative Interviews: Understanding Declarative Visualization Design

To better understand the debugging needs of end users for reactive data visualization, we conducted formative interviews with expert Vega users regarding their development processes. At the time of the study (in May 2015), Vega’s reactive extensions had not yet been officially released, so participants were primarily familiar with static visualizations.

Participants. We recruited 8 software professionals (all male), all with prior experience creating static Vega visualizations. None of the participants were affiliated with the University of Washington. Participants were selected based on their participation in the Vega community. Each interview lasted about 30 minutes; participants did not receive compensation.

Data Collection. The interviews took place over Skype and Google Hangouts. The example visualizations were shared using Google Docs and are included in Appendix C.2. We captured audio recordings for later review and transcribed notes during the interview.

Protocol. The semi-structured interviews examined each participant’s development process as related to Vega. Participants were shown sample visualizations of Vega’s data flow graph (see Appendix C.2) and asked to reflect on the utility of such techniques with respect to their debugging needs; one participant was unable to access and view the sample visualizations during the interview. The full script used for these interviews is included in Appendix C.1.

Analysis. After the interviews, we reviewed the conversations to extract common themes regarding the challenges or potential areas for future work described by the participants. These themes are discussed in more detail in the following section.

5.4.1 Understanding Debugging Challenges and Needs in Vega

Encoding errors are often visually salient (e.g., points are filled with the wrong color), but tracing the error through the specification can be difficult: is the result due to an incorrect scale definition, an error in the data transformations, or a problem with the input data itself?

With Vega’s declarative model, users lack visibility into the state of these components in the underlying system. One participant noted that *“when you mess up that JSON you get an error from deep in JavaScript land”* (P7). In the current development environment, users lack clear approaches to debugging the internal functionality due to several degrees of separation between the user-authored code and the errors that arise for ill-formed specifications. This challenge proves even more difficult when no clear error is communicated by the system, but rather the system output fails to meet the user’s expectations. Another participant described this type of difficult debugging scenario where *“[the resultant visualization is] just blank and you don’t know why”* (P2). Without a clear starting point for the debugging process or tools to facilitate this type of analysis, users may resort to simply reading the specification or to using other primitive forms of debugging, such as iteratively removing or modifying the code.

The proposed data flow graph visualization sought to provide a new method for inspecting the underlying behavior to support users’ debugging needs. However, participants noted that visualizing the internal data flow graph could be beneficial for Vega *system* developers, but provides too much internal information tangential to their *user-level* debugging tasks. In particular, one participant noted that *“the [data flow] graph presumes insight into how Vega’s internals operate”* (P1). Inspecting the state via the JavaScript console (see Appendix B) or viewing Vega’s data flow graph presents users with a mixture of state information, only a small fraction of which is relevant to the debugging task at hand. The extraneous system details complicate identification of relevant information, suggesting that it would be beneficial to strip internal system information from the user’s view. While the data flow graph may not be ideal for debugging particular Vega specifications, the structure can still provide useful insights into the underlying system behavior. In July 2018, Jeffrey Heer published an observable notebook explaining “How Vega Works” [76], which describes the reactive data flow architecture using illustrative and annotated data flow graphs. As a precursor to the descriptive content, Heer notes that *“This notebook assumes basic familiarity with Vega.”* This disclaimer further reiterates the concerns expressed by our earlier formative interview participants about the utility of the data flow graph for general debugging needs.

Given that the data flow graph itself may not be ideal for end-user debugging tasks, the interviews further sought to identify areas in which new program understanding techniques would be particularly helpful. Participants explained that their needs centered on the relationships between data and encodings expressed within their Vega specifications. One participant explained that Vega *“need[s] a way to examine internal variables... [and] to see the internals of the step-by-step process”* (P3). Many user-authored data transformations may restructure the data or introduce new attributes of which users are unaware. Because of these opaque transformations, many participants expressed the need to understand *“the structure of the data that Vega is actually using”* (P6). For the current development environment, one participant explained that *“the easiest path to solve [a specification error] was to just break into the [JavaScript] debugger and see what state the data was in at various stages”* (P3). However, as previously discussed, this particular debugging strategy presents its own challenges and may generally be out of reach for novice Vega users (see Appendix B).

Interactions further complicate the debugging process. For interactive specifications, signals parameterize data transformations and encodings, introducing additional dependencies. Users are then required to reason about the behavior of both the data and interactive signals, which may be interleaved to produce complex interactive functionalities. Furthermore, while signals usefully abstract low-level input events, some users found that this abstraction complicated reasoning about event propagation. As one participant stated, *“debugging reactivity is like a true true nightmare”* (P5). In static visualizations, users already found it difficult to reason about the step-by-step process employed to produce the output Vega visualization; for interactive specifications, users must further understand the time-varying behavior of this pipeline, which adds yet another dimension to the program understanding process.

These interviews illustrate aspects of the three core challenges explored in this dissertation. To better support end-user debugging tasks, systems must communicate details of the internal behavior at the level of abstraction with which users are already familiar. Furthermore, these details should aim to support users in understanding the system functionality as related to their primary development tasks, rather than tangential details about the behavior.

5.5 Summary of Contributions

Declarative languages can enable users to focus on relevant design decisions—such as how interactions should parameterize a visualization—while relying on the system to capture and track the interactive behavior. However, when interactions produce erroneous results, existing debugging techniques such as breakpoints or stack traces are no longer effective since users are unfamiliar with the underlying control flow. In this chapter, we discuss the design of Vega [177] as a platform on which to explore new debugging techniques for reactive programming languages such as React [93] or Elm [36]. Regardless of programming style, interactions can be inherently difficult to author and debug. End-user programmers must understand complex dependencies among input events, program state, and visual output.

In this chapter, we first contribute a data flow graph visualization to illustrate Vega’s complex underlying control flow. To better understand the potential utility of this approach, we then contribute a series of formative interviews with expert Vega users. While participants felt that the data flow graph could facilitate program understanding for Vega *system* developers, the visualization provided too much information tangential to their *end-user* debugging tasks. In this case, the data flow graph did not communicate information at the level of abstraction with which end users were most familiar. Instead, participants felt that new debugging techniques could better help users interpret the time-varying behavior of data transformations and help track changes through visual encoding pipelines. Vega’s well-defined semantics provide new opportunities for enhanced debugging support, as new tools can surface traces from pixels, through scale transforms, to source data (and vice versa). The next chapter continues to explore this space through the development of new visual debugging techniques that better reflect the expertise of end-user programmers in Vega.

Vega was presented at VIS 2015 [177], in collaboration with Arvind Satyanarayan, Ryan Russell, and Jeffrey Heer. Vega’s data flow graph visualization was presented at EuroRV3 2015 [86], in collaboration with Arvind Satyanarayan and Jeffrey Heer. The formative interviews were published as part of our later work on visual debugging techniques at EuroVis 2016 [87].

Chapter 6

VISUAL DEBUGGING TECHNIQUES FOR REACTIVE DATA VISUALIZATION

Vega supports the design of interactive visualizations by allowing the end-user programmer to focus on visual encoding decisions while deferring the low-level implementation details to the underlying system. However, when errors arise in the interactive behavior, the separation between the code the user writes and the output can complicate the program understanding process (see Chapter 5). In formative interviews with visualization developers (Section 5.4), one participant succinctly noted that *“debugging reactivity is like a true true nightmare.”* To help end users understand and debug the time-varying behavior, participants reflected that new systems should help users *“to see the internals of the step-by-step process.”* While the proposed data flow graph visualization provides an accurate portrayal of the underlying Vega control flow (Section 5.3), this approach provides too much information tangential to end-users’ debugging needs. For new program understanding techniques, it is therefore essential that the approach reflect the level of abstraction with which users are already familiar.

To this end, this chapter contributes a set of visual debugging techniques for reactive data visualization motivated by prior work on program visualization and debugging (Section 2.3). To inform the design of the proposed techniques, we identify three design goals: new systems should enable users to **(1) probe the state**, **(2) visualize relationships**, and **(3) inspect transitions**. Vega’s well-defined semantics provide a clear entry point for snapshotting the time-varying behavior, and Vega further acts as a representative platform on which to explore new debugging techniques for a more general class of reactive programming languages such as React [93] and Elm [36]. In this chapter we introduce the design of an interactive timeline to visualize the behavior of signals, in situ annotations to help users interpret the behavior

of visual encodings, and dynamic data tables that visualize the time-varying behavior of the backing data. This chapter further contributes an evaluation with 12 first-time Vega users to examine how users debug faulty interactions in unfamiliar specifications. Despite their lack of expertise with Vega, we find that participants can accurately trace errors to problematic lines in the specifications by employing our visual debugging techniques.

The techniques introduced in this chapter illustrate the importance of communicating relevant system details to end users at the level of abstraction with which they are familiar. By visualizing the core Vega constructs that are represented in the code, users can map the results of the visual debugging techniques to corresponding code elements with minimal expertise in the language. An important aspect of this approach was deciding which low-level system details not to represent in our visual debugging techniques. Rather than trying to completely and accurately represent the underlying system behavior, each component instead focuses on a subset of Vega constructs specifically targeted towards end-user debugging tasks.

6.1 Design of Visual Debugging Techniques for Program Understanding

Based on our formative interviews with Vega users (described in Section 5.4), we identify three core design goals for debugging interactive visualization code in Vega:

1. **Probe the state:** At any given moment, the visualization is determined by signal values, data transformations, and encoding rules. Users must be able to inspect the state of each of these components to better understand the behavior.
2. **Visualize relationships:** The state of one component often affects others—for example, signals can parameterize encoding rules, or data transformations may affect scale domains. Users must be able to identify dependencies between components.
3. **Inspect state transitions:** Input events trigger transitions from one state to another, and debugging faulty interactions requires understanding the causes and consequences of these transitions. To identify the source of an error, users must be able to inspect how values propagate through the user-authored specification.

6.1.1 System Overview

We now present the design of our visual debugging techniques for reactive data visualization. Our debugging techniques were implemented as part of the online Vega editor. In the online editor, there are separate panels to view the user-authored specification and the output Vega visualization. We add additional panels and interactions to this environment to support our visual debugging techniques (Figure 6.1). To enable inspection of the state and the behavior of changes over time, we incorporate three elements: a *timeline* of interactive signals, a *tooltip annotation* showing context dependent visual encodings on the output visualizations, and a *dynamic data table*. The end-user programmer can inspect the underlying program state and *replay* to past states to view the behavior over time. In the following sections, we describe the design and backing rationale for each of these debugging techniques.

In the formative studies (see Section 5.4), one participant observed that “*There are two possible errors. One is like a runtime error... The other is you actually have a well-formed execution and [the visualization] is not showing what you expect it to show*” (P1). These debugging techniques focus on the latter debugging scenario in which the Vega specification is technically correct, but the output Vega visualization deviates from the user’s expectations. The proposed techniques therefore aim to support the refinement of the user’s mental model through exploration of both the data and program state. These techniques focus on the Vega components with which users are most likely to be familiar when developing an interactive visualization: the interactive signals, the backing dataset, and the visual encoding decisions.

We first provide a short, illustrative example of how our visual debugging techniques can be used to better understand and debug the behavior of an interactive index chart. Prior to this work, the online Vega editor did not have debugging support for end-user programmers. Instead, the historical debugging approach for this example only leverages the JavaScript console in the browser. This approach requires users to have additional insight into the Vega system internals that most end-user programmers are unlikely to possess. The complete historical debugging approach for the interactive index chart is described in Appendix B.

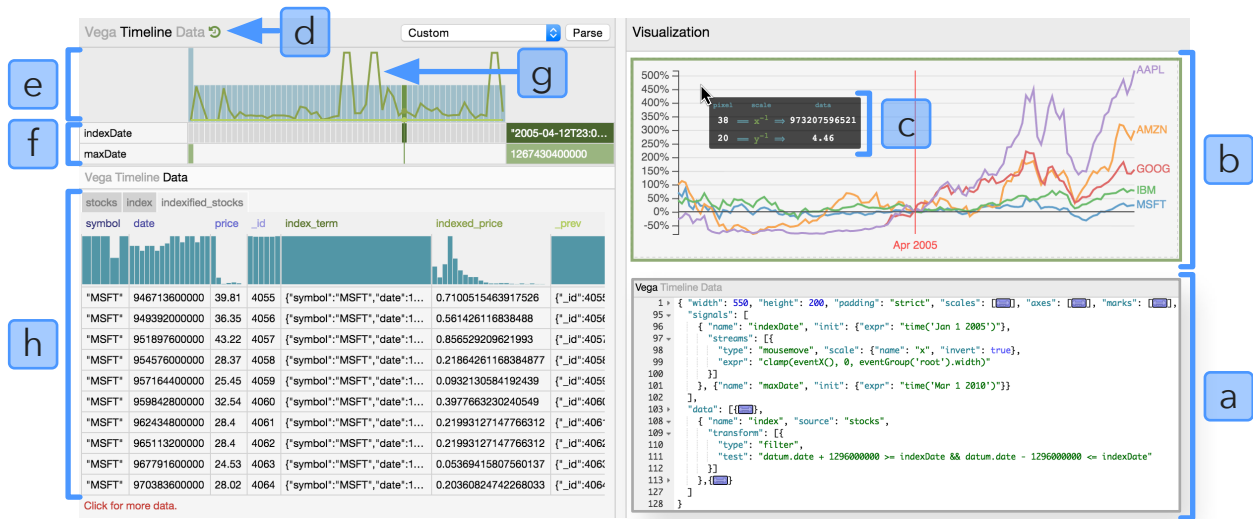


Figure 6.1: Visual debugging techniques for an interactive index chart visualization in Vega. End-user programmer author (a) a declarative specification to produce (b) an interactive visualization. (c) Tooltips on the visualization provide introspection into visual encodings while viewing a past state via (d) replay. Recorded interactions are displayed in (e) an overview and (f) a timeline. In the overview, (g) a time series shows the variability of data attributes in (h) the backing datasets.


Usage Scenario. Consider debugging an index chart of stock prices that interactively renormalizes the data relative to the date associated with the mouse position (Figure 6.1b). The user first writes a specification (Figure 6.1a) of encoding rules and interactions. During interaction, the user notices that at certain time points, all the time series erroneously flatline due to a specification error. The user must now assess the dependencies between interaction, program state, and visual output. The user can start by recording interactions in the *timeline* (Figure 6.1f), and *replaying* (Figure 6.1d) the interaction to observe how events propagate. Mousing over events in the timeline shows the *dependencies* of the signal. The *overview* (Figure 6.1e) summarizes activity, allowing for quick identification of interaction patterns. A *tooltip annotation* (Figure 6.1c) exposes the position encoding by showing the data values and encodings corresponding to the selected pixel. The user can then inspect the backing dataset via *dynamic tables* (Figure 6.1h). Guided by the *attribute variability* (Figure 6.1g), the user observes that some data attributes have been zeroed out. The user can then select the data attribute on the table to link back to the specification to fix the error.



Figure 6.2: The overview, timeline, and signal annotations after performing interactions on a Vega visualization. (a) The overview provides insight into different interaction patterns. (b) Stepping within a pulse allows users to see intermediate states of an interaction. For example, the second scatterplot shows a brush representing the *new brush_start* and *old brush_end*. (c) Dependencies are shown as red outlines in the timeline on hover. (d) Signal annotations overlay the visualization, with fill color encoding temporality: from darkest (past), through red (current), to lightest (future).

6.1.2 The Signal Timeline and Replay

The *timeline* (Figure 6.1f) lists every user-defined signal in specification order. Signal updates are represented as colored cells, arranged into columns corresponding to reactive updates (pulses). The current signal value is displayed on the far right; mouse hover expands the contents and displays any scale transforms used to define the signal. As users interact with the visualization, signal values update and populate new columns in the timeline. By default, cell widths are automatically adjusted so all pulses are visible. An *overview* (Figure 6.1e) summarizes pulse activity over time, with bar heights encoding the number of signal updates on a given pulse. The overview exposes patterns in the recorded interaction (Figure 6.2a), and brushing zooms the timeline to show only pulses within the selected range.

Hovering over a cell displays a tooltip of the signal value in the overview to enable rapid comparison of values. Hovering also exposes the dependencies a signal update relies on—cells are outlined in red to illustrate which dependency values are used, and link icons  are shown beside dependency names in case the corresponding cell is not visible (Figure 6.2c). Keyboard navigation allows users to move up and down to understand the propagation of signal values within the same pulse (Figure 6.2b), or left and right to identify a particular

pulse which exhibited the faulty behavior. The selected cell is shown in **dark green**, and other signal values used by this particular state are shown in **light green**. Users can select a cell in the timeline to *rewind* the visualization to an earlier state. Each time user interaction triggers a signal to update, the system records the new value and pulse number. Replay is enabled by setting the signal values for the desired pulse and re-rendering the visualization. During replay, interaction is disabled to prevent new events from being added mid-stream.

Rationale. The timeline provides users introspection into the heart of the interaction logic—signals—and is designed to reify the two-step reactive update process. As a result, pulses populate the timeline from top to bottom, and hovering over a particular cell reveals if an older value was used for a dependency listed later. Early prototypes took this one step further: pulse propagation was more salient as each cell in the timeline was marginally offset, producing a “cascade” or “waterfall” effect. This design required more space to encode the same information and made coarse navigation difficult. In other words, it was only meaningful to navigate left or right (i.e., backwards or forwards in time). As a result, locating a faulty pulse required users to step through every intermediate state of other pulses. In contrast, by condensing pulses into columns, users can quickly move back and forth across the timeline and only deep dive into the intermediate states of pulses of interest.

The timeline also maintains the level of abstraction provided by signals. For example, the particular low-level input events that trigger a reactive update are not identified. When such low-level events are required for debugging erroneous event selectors, users can define additional signals as needed that only capture the `event.type` that triggers them. Users can then track these changes via the timeline and overview. Similarly, although Vega’s internal data flow dependency graph can be readily visualized (see Section 5.3), the timeline only surfaces dependency information for the particular cell a user hovers over. Helper signals automatically generated by Vega are hidden from the view. Together, these design decisions reflect the findings of our formative study (Section 5.4): users were overwhelmed by details of Vega’s execution pipeline, and found them to be tangential to their primary debugging tasks.

6.1.3 *In Situ Annotations*

When users pause interaction recording, either explicitly (Figure 6.1d) or by *rewinding* to an earlier state, a number of on-demand annotations become available to inspect the visualization state in situ. The specification is analyzed to extract all scaled visual encoding rules for each mark. Mousing over the visualization performs a hit test against the underlying scene graph to find an intersecting mark or group. If a mark is not found, then the user’s cursor is over a group’s background; the tooltip displays the cursor’s coordinates relative to the group, along with any spatial scales used to encode the group’s children (Figure 6.1c). If a mark is found, its visual encoding rules are shown in addition to the coordinates.

Mousing over a signal value in the timeline that duck-types to coordinates (i.e., an object with x and y properties), displays all signal updates as *signal annotations* on the output visualization. The current point is denoted with a white stroke, and the fill color encodes the relative temporality—older points are darker and lighter points occur further in the future (Figure 6.2d). By default, signal annotations are only shown when hovering over the timeline; however, users can opt to have them drawn in real-time as interactions are recorded.

Rationale. Scale transforms are a common visual encoding operation, but can grow complex under a nested scene graph model such as Vega’s. For example, scales defined within nested group marks can shadow scales with the same name at higher levels. Generalizing an interaction technique requires invoking an inverse scale transform, to move from pixel to data values, but identifying the correct scale to use can be error-prone. Vega’s scene graph can be easily visualized but would still require a user to manually map its tree structure to the resultant visualization. Instead, our in situ annotations make inspection of the scene graph a direct manipulation operation. So as not to conflict with user-defined interactions, the annotations only appear when interaction recording is paused.

6.1.4 *Dynamic Data Tables*

Dynamic data tables (Figure 6.1h) display each user-defined dataset. The data tables provide users with a rapid, high-level sense of the backing data. Tables initially show only the first

ten rows, which can be extended on-demand. This sample data allows users to review the attributes of each dataset. Histograms summarize the distribution of each attribute at the given timestamp. Selecting a bar highlights corresponding values in the data table. The data tables update automatically as the user interacts with the visualization to immediately depict changes in the distributions of data properties. While inspecting the table, a time series of the *variability* of each property is shown in the overview (Figure 6.1g). The variability can help users identify particular points during the recorded interaction that caused large changes to the underlying datasets. Mousing over the name of an attribute shows only the corresponding time series. The variability is calculated as follows, where bin'_i is the number of values in bin i of the histogram at the current state and bin_i is for the previous state: $\sum_{i \in bins} |bin'_i - bin_i|$. The variability for static attributes is a flat line along the bottom of the overview.

Rationale. In the formative study, one participant noted that users “*have this expectation about data... [that] is kind of unspoken and pretty hard to debug*” (P7). Interaction undoubtedly exacerbates this problem, as signals can further parameterize data transforms. By displaying the resulting data values for each dataset (i.e., after transforms have been evaluated), our dynamic data tables narrow the gulf of evaluation [92]. Moreover, the overview is augmented with the dataset variability to help users map the effect of signal updates to changes in the datasets. The current calculation for the variability detects large shifts in the distribution of data, instead of individual property values, in order to better highlight surprising changes. As with the timeline, datasets internal to Vega are hidden from the view.

6.1.5 *Linked Highlighting of the Specification*

Users can select the name of a signal or data property in order to highlight all occurrences of that name in the specification; this functionality allows users to better connect the behavior seen in our visual debugging techniques back to the relevant context in the code. If the specification panel is not currently visible in the development environment, it will be displayed alongside the current view. This linking allows users to rapidly trace variables from the timeline or dynamic data tables back to the original specification.

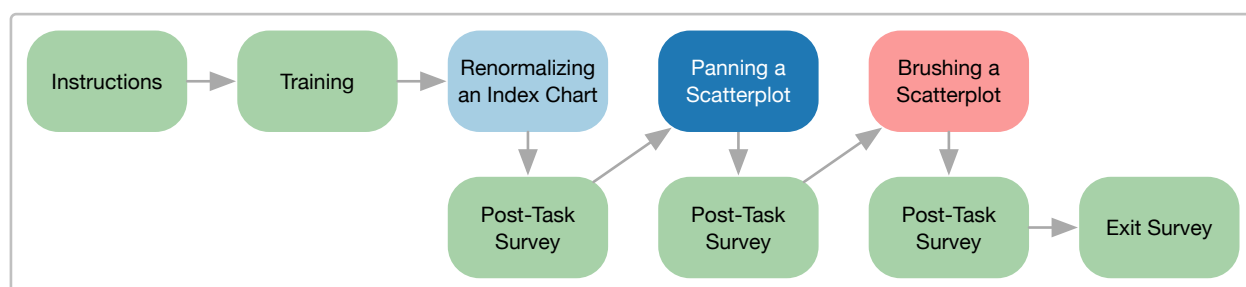


Figure 6.3: Participants completed three evaluation tasks, each followed by a post-task survey to identify candidate lines as the source of the error and to rate the visual debugging techniques.

6.2 Evaluation: Debugging Faulty Visualizations

We conducted a study of how 12 first-time Vega users employ these debugging techniques to assess faulty specifications. Across a set of real-world errors, we examined participants’ debugging strategies and their interactions with our visual debugging techniques.

Participants. We recruited 12 first-time Vega users (8 male, 4 female), all with prior experience analyzing data and creating visualizations. Participant ages ranged from 23 to 42 (mean 27.5, s.d. 5.14). All participants were either graduate (11) or postdoctoral (1) students at the University of Washington. Each study session lasted about 90 minutes; each participant received a \$15 gift card as compensation for participation in the study.

Protocol. Prior to the study, we asked participants to review Vega’s beginner tutorial [196]. We began the study with another Vega tutorial to introduce the visual debugging techniques. Participants were also provided with a reference sheet containing the names and descriptions of each technique (Appendix D.2). At the start of each task, we oriented participants with an explanation of the visualization and its intended functionality. Participants were then asked to diagnose the behavior by identifying one or more lines in the code that cause the error. As part of the post-task survey, participants rated the utility of each visual debugging technique. At the end of the study, participants completed an exit survey. The methodology for the study is shown in Figure 6.3. The survey questions are included in Appendix D.1.

Participants completed three tasks, each with an unfamiliar specification. Each specification was based on a real-world error encountered by Vega users and developers. These errors

represent a range of breakdowns, covering data transformations, interaction logic, and visual encodings, respectively. Tasks were ordered by increasing conceptual difficulty and emphasize different parts of the system. Detailed descriptions are provided in the following sections.

We used existing specifications rather than requiring participants to craft visualizations from scratch in order to focus the evaluation on known debugging challenges. This approach also ensured that each participant encountered the same set of errors to facilitate comparisons. Participants' unfamiliarity with Vega provided a conservative test of our debugging techniques, as participants could not rely on prior experience to inform the debugging process. As described in Section 5.4, the previous debugging strategy required user familiarity with the Vega system internals, which is not otherwise necessary when authoring visualizations. Given that most users lack this familiarity, the previous debugging process is not representative of the behavior of real-world users and is not a fair comparison for our expected use case. Manual exploration of the internal Vega structure is more low level than the abstraction used when writing specifications and thus less fit for general debugging scenarios. In particular, the historical debugging approach relies heavily on the use of the JavaScript console to navigate the system internals. A demonstration of this debugging approach is included in Appendix B. Future work is required to assess how these visual debugging techniques will be employed in real-world development processes by expert users.

Data Collection. We used a think-aloud protocol throughout the study. Audio and screen recordings were captured for later review. At the end of each task, participants completed a brief survey in which they identified faulty lines of the specification and explained the reasoning for their choice. Participants also provided Likert ratings of the usefulness of each debugging technique. At the end of the study, participants ranked the debugging features and provided written impressions of the debugging experience overall (see Appendix D.1).

Analysis. We assessed participant accuracy by checking if they correctly identified lines in the specification related to the error. We examined the average rating of each technique for each task and assessed the utility of techniques for different types of errors.

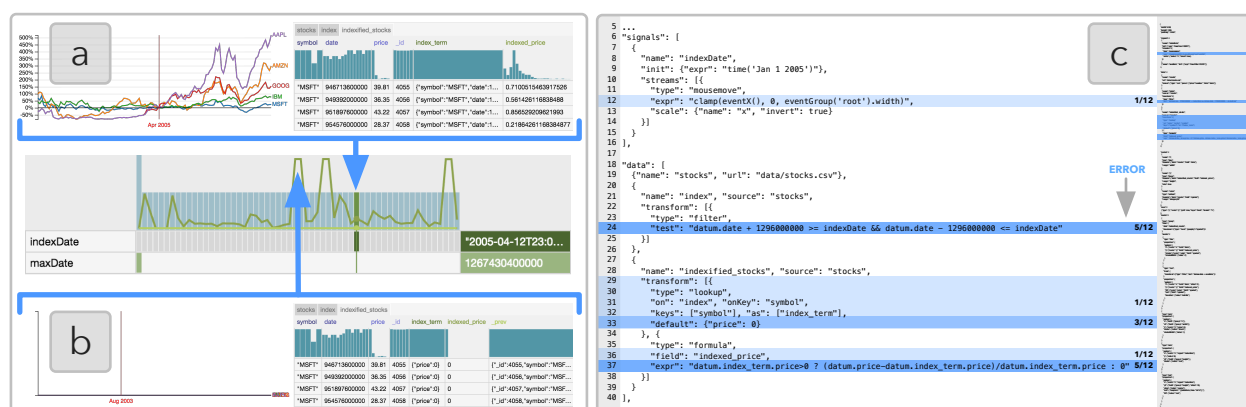


Figure 6.4: (a) An index chart interactively renormalizes the stock price time series data based on the mouse position of the interactive cursor, (b) but a data transformation error sometimes zeros out the `indexed_price` variable, causing the chart to flatline. (c) An excerpt of the specification shows the distribution of lines identified by participants as the source of the error (Line 24).

6.2.1 Data Transformation Errors: Renormalizing an Index Chart

An index chart of stock prices normalizes the data relative to a mouse-selected time point. At certain dates in the visualization, the lines flatline due to an erroneous data transformation that incorrectly filters the backing dataset (Figure 6.4b). The filter uses a constant to specify a range with the same month and year as the index point, but the constant incorrectly excludes some points due to a slight time offset between the data and index point. The error can be fixed using Vega's date support to compare the month and year. Appendix B describes this error using the JavaScript console. Participants had 15 minutes for this task.

Five participants (42%) correctly identified the exact line causing the error. All remaining participants correctly identified dependent lines that are corrupted by the faulty data filter (Figure 6.4c). Participants identified nine distinct lines (out of 145). Participants first identified dates where the visualization flatlines. By replaying to those points in the timeline, all participants verified that the signal value for the index point was capturing a logical date. Participants switched to the data table to compare attributes across states and observed that during the error condition, the `indexed_price` was always zero. Participants linked back to the code to identify dependencies and select candidate lines. Participants rated *replay* and the *data table* most highly (Figure 6.7 ●). The data table is essential for identifying corrupted values from the faulty filter transformation. Replay is crucial for isolating the error states.

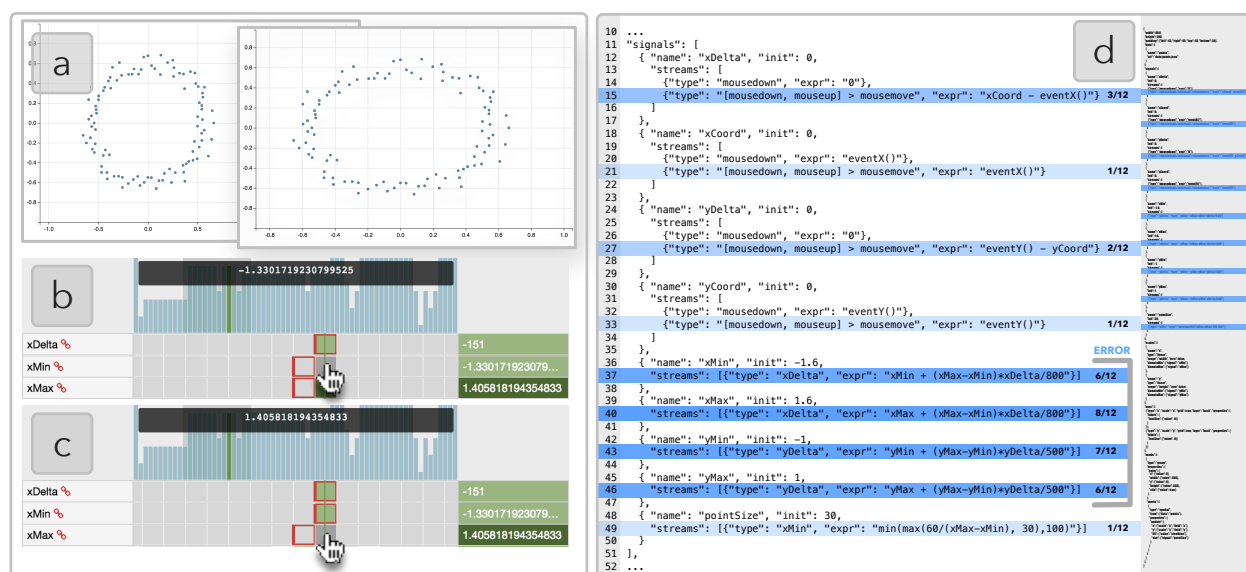


Figure 6.5: (a) As the user pans the scatterplot, the axes begin to stretch and distort the plot. This distortion occurs due to an interdependency in the definition of the signals responsible for setting the range of the scale. (b) The `xMin` signal uses the old values of both `xMin` and `xMax` to compute its new value, whereas (c) the `xMax` signal uses the new `xMin` value and the old `xMax` value, thus causing the range to drift. (d) An excerpt of the specification indicates the problematic lines and shows the distribution of lines identified by participants as the source of the error.

6.2.2 Interaction Logic Errors: Panning a Scatterplot

A scatterplot supports panning via mouse drag. Over repeated panning actions, the aspect ratio of the plot distorts (Figure 6.5a). Panning is implemented as a set of signals defining the minimum and maximum domain values for each axis (`xMin`, `xMax`, `yMin`, `yMax`). The error occurs due to a mutual dependency between these signals: the minimum signal uses the old minimum and maximum values to compute the new value (Figure 6.5b), whereas the maximum signal uses the *new* minimum value and *old* maximum value (Figure 6.5c). Resolving the error requires a redesign of the specification to remove the mutual dependency in the interaction logic. Participants had 20 minutes for this task.

Eight participants (67%) correctly identified the minimum and maximum signals as the source of the error. One participant even attempted to fix the error. The remaining participants identified either immediate upstream or downstream dependencies of the erroneous signals (Figure 6.5d). Participants identified nine distinct lines (out of 136). Participants

started the debugging process by panning the scatterplot and forming hypotheses about the behavior of the error. In testing each hypothesis, participants often reset the timeline to only view the most recent signal updates. Once participants observed the distortion in the scatterplot, they used the timeline to compare the signal behaviors. To assess the relationships between signals, some participants used the *dependency* markers to determine how the signal values propagated whereas others attempted to glean these relationships from the specification. Due to participants' lack of familiarity with the Vega syntax, reading the specification alone in the short time frame was a challenge. Participants noticed that many signals computed the difference between the minimum and maximum to represent the visual range, and noted that the size of this range should not be changing during the panning interaction. Participants thus identified signals utilizing this computation as candidate lines. Participants rated the *dependencies* and *timeline* most highly for this task, as they revealed the relationships between signal values and the underlying interaction logic (Figure 6.7 +).

6.2.3 Visual Encoding Errors: Brushing a Scatterplot

A scatterplot enables brushing to highlight points: points within the brush extents should have their fill color updated. The pixel values of the brush extents are run through scale inversions to determine a selection over data attributes. However, the brushing interaction does not always highlight points when the visualization is first parsed (Figure 6.6a). The error occurs because the `scatterplot` signal (which represents a group mark containing the plot) is needed to find the appropriate scale to invert the pixel-level brush extents, but is initialized as an empty object that is only set on `mousedown` events. However, these events do not correctly propagate if the user performs a `mousedown` on the background, as the enclosing group element has no fill color (an idiosyncrasy inherited from Scalable Vector Graphics). If the `mousedown` occurs over any of the plotting symbols, which do have a fill color, the event fires and `scatterplot` is accordingly set (Figure 6.6b), enabling all future brushing actions to work appropriately (Figure 6.6c). This example is a simplification of a breakdown that can occur in scatterplot matrices. Participants had 15 minutes for this task.

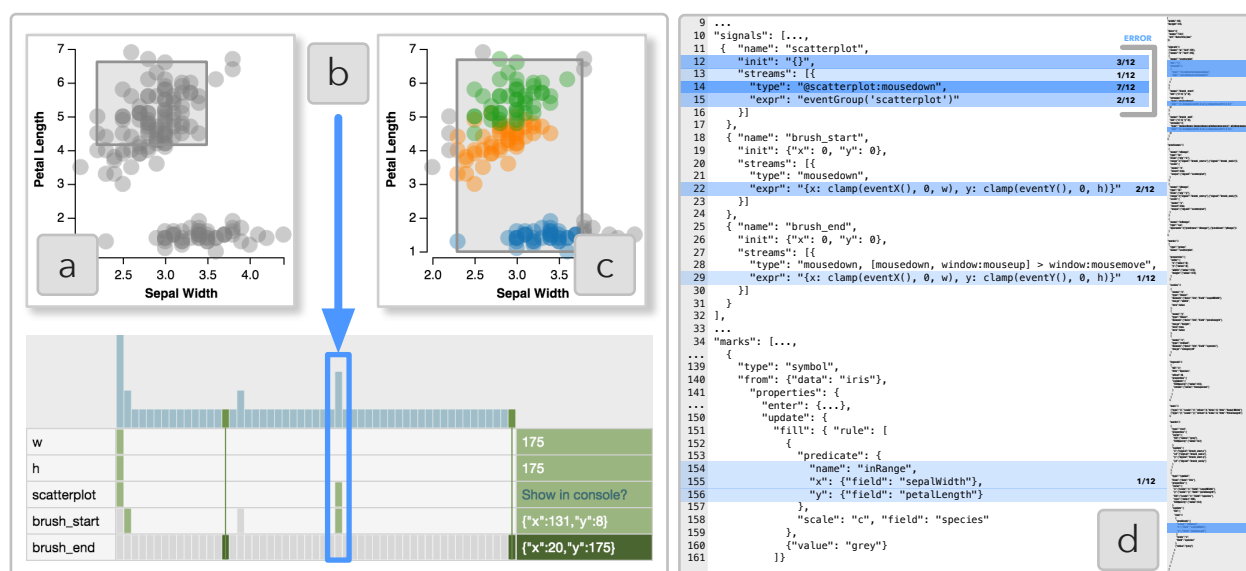


Figure 6.6: Relevant scale definitions are extracted from the `scatterplot` signal to drive the color encoding of the brush. The `scatterplot` signal is initialized as an empty object, which causes (a) the brush to display but fails to highlight the points. When a point is clicked, (b) the `scatterplot` signal is defined and (c) the brush works correctly for all future interactions. (d) An excerpt of the specifications displays the distribution of lines identified as the source of the error.

Nine participants (75%) correctly identified the `scatterplot` signal as the source of the error (Figure 6.6d). The remaining participants incorrectly selected lines associated with the brush signal and the fill color encoding. Participants identified eleven distinct lines from the specification (out of 176). Two participants implemented a partial fix by changing the definition of the `scatterplot` signal to update on `mouseover` instead of `mousedown`. While this solution causes the brush to correctly color points, it does not correctly address the problem of event propagation as described above. Participants began by trying to reliably reproduce the erratic brushing behavior. Once the conditions of the behavior were determined, participants examined the timeline to compare the signals across working and faulty brushing runs. Participants observed that when a mark was selected, the `scatterplot` signal was set in the timeline to the appropriate scope. By selecting the `scatterplot` signal in the timeline, users highlighted its use in the specification in order to identify the corresponding specification lines. Consequently, the *timeline* received the highest ratings (Figure 6.7 ■), since it allowed participants to observe and track the inconsistent behavior of the `scatterplot` signal.

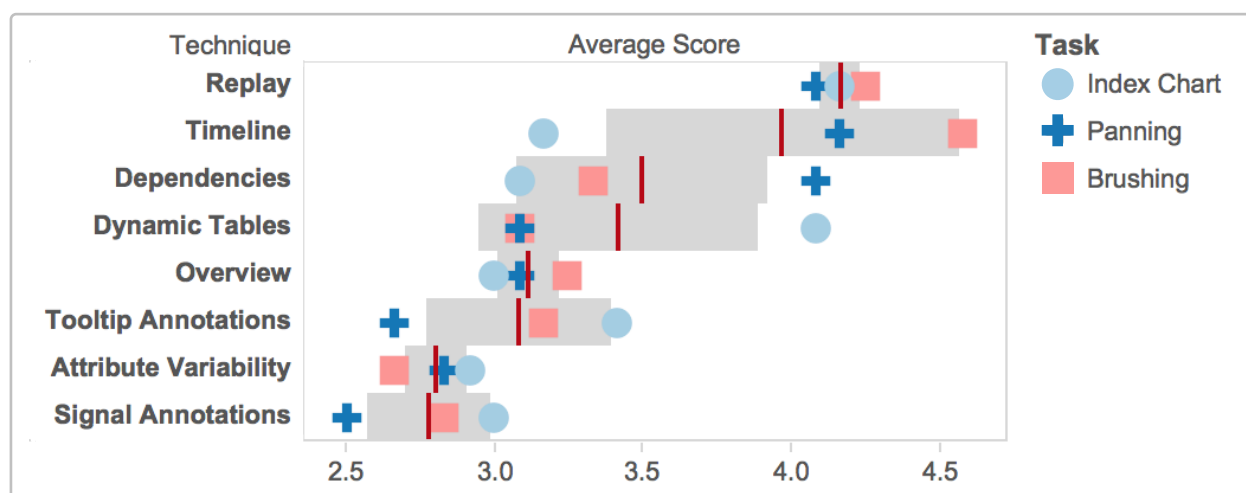


Figure 6.7: Average ratings for the utility of each debugging technique, based on the task (shape) and across all tasks (lines), with one standard deviation shown in gray.

6.2.4 Discussion of Evaluation Results

For each task, the majority of participants were successful in either precisely identifying erroneous specification lines or detecting lines directly related to the error. Despite being first-time users, participants accurately identified erroneous lines for faulty panning (67%) and brushing (75%) interactions. Three participants even attempted partial fixes (1 panning, 2 brushing)—an encouraging result given their lack of familiarity with Vega. In only 15-20 minutes, these participants were able to observe, diagnose, and experiment with solutions to the error in an unfamiliar specification and environment. For the index chart, 42% of participants correctly identified the problematic line, with the remaining participants identifying dependent lines corrupted by the error. As participants used the debugging techniques to conceptually hone in on an unfamiliar problem, we consider this a promising result.

Figure 6.7 plots participant ratings for each debugging technique. Of particular note is that the utility of each technique is highly dependent on the type of error—for example, *dynamic data tables* were rated highly for the index chart, which featured a data transformation error, whereas the *timeline* was rated poorly as the interaction for the index chart required only a single signal. In order to understand the complex dependencies within the panning example, the *dependencies* on the timeline were much more salient. On average, the

combination of *timeline* and *replay* techniques were deemed universally useful for assessing program state and observing relevant changes (Figure 6.7). One participant noted that “*the combination of the timeline, replay, automatic text highlight, and dependencies makes for a pretty useful and smooth debugging experience.*”

The remaining techniques (*overview*, *tooltip*, *attribute variability*, and *signal annotations*) were rated lower on average as each technique was less effective at surfacing information relevant to the debugging tasks. The *tooltip* had the largest spread of average ratings, and was particularly useful in debugging the index chart by allowing users to inspect the encoding of the broken state and track the underlying error to the backing dataset. The *attribute variability* was designed to support quick identification of data changes, but was often overlooked by participants. One participant noted that the low rating suggests that the system should “*promote its appearance more.*” Currently, users must explicitly select the debugging technique they wish to use, which requires them to know what information would be most useful. Future development of these techniques should examine how to automatically surface relevant details with less intervention; we further explore this idea in Chapter 7. Additional static analysis, or new higher-level specifications, could help the system better understand the semantics of interactions (e.g., do signals define point or range selections?) and automatically surface appropriate techniques.

6.3 Limitations and Future Work

Vega’s reactive semantics enabled us to efficiently snapshot the program state at every point in the execution history; furthermore, we were able to utilize the structure of the Vega code to easily identify and visualize all relevant program variables. While this work exemplifies how the techniques may be applied for a large class of reactive programming languages, future work should explore how to effectively incorporate real-time program visualizations into imperative programming domains. Extending this work to such programming contexts requires additional insight into how to efficiently log relevant details of the program execution [26]. While the semantics of Vega facilitate identification of relevant program variables, future

work would need to explore new workflows to predict which variables are most relevant to end-user debugging tasks. One strategy would be to employ static analysis of the program behavior prior to executing the code or to require users to manually annotate interesting parts of the program execution [36]. However, future work is needed to explore new ways to limit the burden on programmers to better support preventative debugging strategies that surface surprising details before the user knows to look for them (see Chapter 7).

The replay technique currently employed in this work updates the visualization by setting the signal values of the previous state and re-rendering the visualization as if it were a new pulse in the execution. However, this functionality assumes a consistent definition of the set of signals, limiting support for hot-swapping changes in the specification. Future work should examine what additional information should be recorded to support replay of interactions across specification changes. Though low-level input events are abstracted into signal definitions for easier debugging by users, such events may be necessary to support replay when signal definitions have changed or been added to the specification.

In the evaluation, one participant explained that *“I would have loved a way to use the visualization essentially as an editor to modify the specification (and then see those changes update the viz in real time).”* This approach for direct manipulation design has been explored through a number of visualization construction systems (see Section 2.1.2). For example, Lyra [174] provides an interactive environment for visualization design via direct manipulation, but does not yet support authoring interactions. Our timeline visualizes the propagation of events to the interaction logic, but may be too low level for an interactive development environment like Lyra. By shifting the focus of the signal annotations from a summary of all events, to an indication of the current state, the annotations could support better debugging of interaction sequences in situ. Replay could then support playback and refinement of interaction sequences to enable authoring of interactions in an interactive design environment like Lyra [174], Data Illustrator [129], or Charticator [161].

6.4 Summary of Contributions

Interaction techniques are crucial for exploring and understanding visualizations, but are often difficult to author and debug. Vega’s reactive semantics encapsulate the bulk of the interaction logic, providing a meaningful entry point for the program understanding process. We contribute a set of visual debugging techniques that allow users to **probe the state**, **visualize relationships**, and **inspect state transitions** over time. These techniques include a timeline visualizing interactive signals, in situ annotations of visual encodings, and dynamic data tables. The three tasks in the user evaluation demonstrate data transformation, interaction logic, and encoding errors that arise during the design of interactive visualizations. The evaluation demonstrates how the proposed techniques can be used by novice users to accurately identify and understand these errors and better support their debugging needs. In particular, these visual debugging techniques surface relevant details of the underlying system behavior at the level of abstraction with which users are most familiar. Using these debugging techniques, three participants felt sufficiently comfortable with the unfamiliar code to attempt to fix the errors rather than simply identifying the source of the error.

From the evaluation, we found that the utility of each technique was highly dependent on the debugging needs and awareness of the participant. For example, the *attribute variability* was designed to support quick identification of data changes, but was often overlooked by participants. In response, one participant felt that the system should “*promote its appearance more.*” For this work, users must explicitly select the debugging technique they wish to use by navigating to the appropriate part of the debugging environment. However, to select the right approach requires the end-user programmer to already know what information would be most useful for a given task. In the next chapter, I present a design space of program visualizations that augment source code and thus reduce the separation between the end-user programmer’s code and the proposed program understanding visualizations.

This work was done in collaboration with Arvind Satyanarayan and Jeffrey Heer, and was originally published and presented at EuroVis 2016 [87].

Chapter 7

AUGMENTING CODE WITH IN SITU VISUALIZATIONS

Similar to the visual debugging techniques described in the previous chapter, many existing program understanding tools must be explicitly invoked and are presented to programmers in separate, coordinated views. However, studies have shown that switching between views imposes a burden on developers, making it difficult for them to maintain a clear picture of the overall context of the runtime behavior [118, 147, 151, 167]. This separation is particularly problematic when programmers are immersed in a particular task—they may overlook details that would be obvious in an alternate view [167]. Expectations about the desired behavior may also cause programmers to overlook errors when the behavior diverges from expectation. These scenarios illustrate the challenge faced by systems to “support the tasks that matter most to the user.” While debugging approaches may be available (and useful) for testing or debugging tasks, these same approaches may be inappropriate when the programmer’s goals shift towards authoring or reviewing the code. New approaches are therefore required to support proactive program debugging by adapting to the user’s changing needs.

To better support proactive program understanding for source code, we explore the design of program visualizations that are displayed directly inline in the code. To this end, we contribute a design space of embedded visualizations for interactive applications that visualize the behavior of time-varying variables. As with the visual debugging techniques introduced in Chapter 6, the in situ visualizations represent the program behavior at the level of abstraction with which users are most familiar. Snapshots of the program state highlight the exact value of scalar variables, or the underlying distributions of set variables. Sequence visualizations demonstrate how these variables change over time. Motivated by prior work [66], we further contribute criteria for the placement of code augmentations based on trade-off

metrics, such as considerations for the amount of code reflow or occlusion, and the impact on the comparability, unobtrusiveness, and salience of the visualizations.

We then show that these interactive visualizations of the program state can enable richer interactions across the development environment and present runtime information as a first-class component of the code authoring process. In an evaluation with 18 first-time Vega users, we found that participants could improve their overall task grade on a set of program understanding questions by about 2 points (out of 42) when using the in situ visualizations. Furthermore, these in situ visualizations helped increase participants' perceptions about the speed and accuracy of their answers and helped better situate themselves within the code.


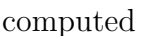




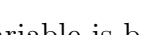
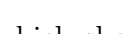
Detail	Type	Temporality	Name	Visualization	Page #
Data: Visualize the exact value or underlying data distributions	Value	Snapshot	Exact Value		100
		Sequence	Line		
			Horizon		
	Set	Snapshot	Heatmap		101
			Histogram		
		Sequence	Summary Line		
			Summary Horizon		
	Change: Visualize how variables update between steps in the execution	Value	Snapshot	Indicator	No Change: Change:
Sequence			Timeline		103
			Tick		
Set		Snapshot	Modification Indicator		104
		Sequence	Stacked Area		

Figure 7.1: We identified ten visualizations designs for code augmentations based on the level of *detail* to show, the data *type* of the variable, and the *temporality* level of interest.

7.1 Design Space of Code-Embedded Visualizations

Programmers must understand the values of, and changes to, variables as their code executes. In this section, we describe a design space of code-embedded visualizations (Figure 7.1). We decompose the design space into two types of visualized data (value and set) at two levels of detail (data and change) across two temporalities (snapshot and sequence). We will first motivate the design of our in situ visualizations with a usage scenario and a set of design considerations. Programmers often have a set of implicit assumptions about how their code will behave, which reflect their original intentions when writing the code.

Consider a scenario in which a programmer is creating a scatterplot that supports panning and zooming. The programmer defines `xMin`, `xMax`, `yMin`, and `yMax` values to track the viewport position and update the variables based on how much the end user has interacted with the visualization. These values can then be used to determine the domain for each axis (`xDomain` and `yDomain`). The programmer also decides to vary the point `size` based on the relative zoom level, using the span of the `xDomain` as a proxy. However, the programmer accidentally introduces an error where the `xMin` and `xMax` values are mutually dependent and therefore cause the `xDomain` to stretch as the end user pans the visualization. With the visual encodings updated to reference these variables, the programmer may begin testing the output via interaction. This example is based on the evaluation task from Section 6.2.2.

Usage Scenario. The programmer starts by performing pan operations to see how the visualization updates based on changes to the axis minimum and maximum values (`xMin` , `xMax` , `yMin` , `yMax` ) , the axis domains computed from these values (`xDomain` , and `yDomain` ) , and the point `size`  . However, even while *only* panning, the scatterplot visualization seems to *also* zoom into the points. This behavior is surprising, as it does not reflect the programmer’s intentions. Looking at the dynamic code behavior indicated by the inline visualizations, the programmer notices that the `size`  is increasing while panning. The `size` variable is based on the span of the `xDomain` (which should not change while panning), revealing an underlying error in how the `xDomain` is computed: the `xMin` and `xMax` values are mutually dependent

and thus produce an error when updated sequentially. The in situ visualizations provide a contextually appropriate representation of the program behavior directly in the code, thus reducing the separation between code authoring and program understanding.

7.1.1 Design Considerations

Code augmentations narrow the gulf of evaluation between the programmer’s code and the runtime behavior by surfacing contextually relevant information in situ. Motivated by prior work, we describe three requirements that inform the design of effective in situ visualizations: code augmentations must be **(1) comparable**, **(2) salient**, and **(3) unobtrusive**.

1. **Comparable:** In situ visualizations help programmers draw connections between variables in the code and the runtime behavior, and should further facilitate identification of important trends. Programmers may need to interact with multiple visualizations to understand relationships between different variables in the code, by using on-demand linking [17], for example. When comparability is essential to the programmer’s current task (e.g., to compare related variables), the augmentations should prioritize placement decisions that facilitate comparison by using alignment and shared axes [154].
2. **Salient:** The code augmentations should adapt to provide contextually relevant information for the programmer’s current task and should update their salience to attract the programmer’s attention to potential areas of interest. The placement [66, 211] and animation [75, 136] for a code augmentation can influence the salience. The *temporality* (snapshot or sequence) of the augmentation impacts its utility for particular tasks.
3. **Unobtrusive:** The code augmentations must remain unobtrusive so as to not detract from the programmer’s primary task. The amount of text *reflow* and *occlusion* [66, 211] can increase the obtrusiveness; the code augmentations should therefore minimize changes to the code position and visibility when the programmer is actively reading the code. However, violating such layout concerns may help maintain the code structure or improve salience at the programmer’s periphery (e.g., while testing the behavior).

```

33 ▾ {
34   "name": "indexified_stocks",
35   "source": "stocks",
36 ▾  "transform": [{
37     "type": "lookup",
38     "on": "index", "onKey": "symbol" ██████,
39     "keys": ["symbol" ██████], "as": ["index_term" ██████],
40     "default": {"price": 0}
41 ▾  }, {
42     "type": "formula",
43     "field": "indexed_price" ██████,
44     "expr": "datum.index_term.price > 0 ? (datum.price - datum.index
45 ▾  )]
46 ▾ }

```

Figure 7.2: The `index_term` variable in this Vega [177] specification represents an array of objects, so we select a representative property to visualize and differentiate the augmentation from others using the **orange** color. On line 44, the programmer uses the `index_term.price`, so we choose `price` as the representative property. The selected object key is also shown on mouseover.

7.1.2 Data Type: Value and Set

We separate program variables into two data types: value and set. Value variables represent a single element of interest to the programmer that takes the form of either a value (e.g., a number, date, or string) or an object (e.g., a set of key-value pairs). Set variables represent a collection of *value* elements for which the programmer needs to understand the underlying distribution. To represent objects in both value and set data, we perform an object simplification in which the object is represented by one of its properties (Figure 7.2). We select a representative property of the object by identifying which of its properties is most commonly used within the code. Visualizations of objects are differentiated from others with an **orange** color to help avoid misunderstandings about the type of object variables.

7.1.3 Level of Detail: Data



For the *data* level of detail, the programmer is interested in understanding the exact value and underlying distributions of the program variables. We thus identified several visualizations to highlight properties of the data at different temporalities.




*Exact Value*LEVEL OF DETAIL: **data**, DATA TYPE: **value**, TEMPORALITY: **snapshot**

When viewing a snapshot of the runtime behavior, we display the exact value `1980` of the data variable. For variables representing a single object, we produce a simplified representation that shows a single property of the object `{ "x": 42, ... }`; the full object can then be viewed on-demand `{ "x": 42, "y": 56 }` via mouseover.

Rationale. The programmer often has expectations about the type or value of variables. Displaying the exact value makes it easy to determine whether or not the value is of the expected type or near the expected value. Development environments often enable this check as an on-demand tooltip showing the current value [9]. Whereas other development environments require the user to view this information via mouseover, our representation reduces the need for interaction by surfacing the same information automatically. Interaction is only required when users want to view the full value of object variables (which may be arbitrarily large).

*Line Chart and Horizon Chart*LEVEL OF DETAIL: **data**, DATA TYPE: **value**, TEMPORALITY: **sequence**



Sequence representations emphasize comparisons across the history (or a subset of the history) of the program runtime. We identified two visualizations to show sequence data: a line chart  and a horizon chart  [79].


Design. For numeric values, we display the exact value over time. For categorical values, we position each category at its own point on the y axis, which allows the programmer to view trends in the visitation history; for example, a sawtooth pattern  indicates habitual revisitation of an earlier state. For arrays of values, we create a line for each element in the array , based on the value type. The length of the history can quickly surpass the number of states that a programmer can easily reason over; in response, we limit the number of states visualized as the program executes to a subset of the most recent states. We found that visualizing up to twenty states with the horizon chart  provides an interpretable view of the data, using four layers (for both positive and negative values) [79]. The programmer can expand the time window on-demand to view a larger slice of the history.

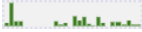
Rationale. The horizon chart is particularly useful for comparisons across positive and negative values [79], whereas the line chart provides a more easily interpretable view of the overall trend. For categorical values, the line chart may be useful for visualizing patterns in the visitation history, but does not otherwise encode useful information in the y position.

Histogram and Heatmap

LEVEL OF DETAIL: **data**, DATA TYPE: **set**, TEMPORALITY: **snapshot**



To provide an overview of set variables, we identified two visualizations to represent the underlying distribution of set data: a histogram  and a heatmap .

Design. For sets of numeric values, we arrange the values into uniformly sized bins. For categorical data, we compute distinct bins for each category and visualize the top n . We place all remaining values in a separate (“other”) bin and visualize it alongside the top n . The “other” bin is colored black and is drawn to scale *up to* the size of the largest bin in the top n . This representation allows the programmer to make comparisons among the largest bins while still representing all the data. For example, in this histogram , the set contains eight different values in varying quantities. We visualize the top n (where $n = 6$) and thus place two values into the “other” bin.


Rationale. We recommend the histogram as the position encoding is more effective than the color encoding in the heatmap [132]. We found that visualizations with a size of about eight pixels per bin are easily interpretable (as in the previous examples shown in this section) and can support interaction on the elements. Representations that reduce the width to about two pixels  make it harder to distinguish between bins or interact effectively. We include the heatmap for its amenability to miniaturization, which we discuss in Section 7.1.5.

Summary Line Chart and Horizon Chart

LEVEL OF DETAIL: **data**, DATA TYPE: **set**, TEMPORALITY: **sequence**

For set variables, the sequence representation aggregates the underlying results to provide an informative summary of the behavior over time. Similar to the *value* type, we visualize the aggregated sequence data as either a line  or horizon chart  [79].

Design. There are multiple ways to represent the value of a set variable at the sequence temporality. For this work, we compute the variance of the dataset and visualize the difference in the variance between the current and previous points in the runtime behavior. Numerous aggregation measures could be applied, and the utility of these measures is highly dependent on the programmer’s task and requirements for the dataset. As such, programmers can configure the system to use the appropriate aggregation measure for their task.


Rationale. Our decision to use the *difference* in the variance was selected to show large shifts in the underlying distribution of the data between states in the program runtime. However, if the difference between states is of less interest, standard aggregations (e.g., mean or median) may be more appropriate. For set representations using the difference measure, we recommend using the horizon chart  as it more strongly emphasizes large values and the direction of the change [79]. Horizon charts also provide a more easily interpretable view of small differences given the small size of the visualization.



7.1.4 Level of Detail: Change

At the *change* level of detail, the *value* and *set* data types are simplified to indicate how variables change. For *value* variables, the *change* is a boolean indicator of whether or not the variable was updated between snapshots of the program runtime. For *set* variables, the *change* is defined as the number of elements that were added, modified, or removed. The *change* level of detail helps attract the programmer’s attention to dynamic variable updates.

Indicator

LEVEL OF DETAIL: **change**, DATA TYPE: **value**, TEMPORALITY: **snapshot**


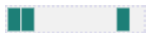
The indicator  shows whether or not a program variable has changed between particular snapshots (or within a certain time period) of the runtime behavior.



Design. The indicator is displayed as either empty  when no change to the variable has occurred, or filled  when the variable has changed between snapshots of the program runtime. The indicator can be extended to provide more information by displaying an arrow indicating the direction of the change or other informative inline glyphs.

Rationale. The indicator acts as a midpoint between the *snapshot* and *sequence* temporalities by showing a comparison of the current and previous states, or by summarizing whether any changes have occurred in a set of states. This augmentation is the simplest one proposed in this work and provides a small indication of where programmers may want to focus their debugging efforts when understanding changes in the runtime behavior.

Tick and Timeline

LEVEL OF DETAIL: **change**, DATA TYPE: **value**, TEMPORALITY: **sequence**


We identified two visualizations to show the history of changes to value variables at the *sequence* temporality: tick  and timeline .

Design. The timeline visualization shows a bar for each state in which the variable is updated. The tick visualization shows a teal block  when a variable is updated and a brown block  for states when the variable is not updated, similar to the indicator visualization.

Rationale. The tick visualization is based on Tufte’s baseball sparkline [192], whereas the timeline is motivated by the timeline we developed in Chapter 6. For the change level of detail, we recommend the tick visualization as it provides a clearer indication of the behavior for every snapshot in the sequence. The tick visualization allows the programmer to inspect the value of a program variable even when it has not been recently updated; for example, the programmer may be interested in the value when it is used but not updated (e.g., the variable represents a previously defined value and is only “read”). The redundant position and color encoding allows the programmer to easily extract the update status at a glance while also facilitating miniaturization (see Section 7.1.5). The timeline visualization may be more appropriate when the variable does not have a value during states when it was not updated (e.g., the variable only exists for states when it has been newly defined).

Modification Indicator

LEVEL OF DETAIL: **change**, DATA TYPE: **set**, TEMPORALITY: **snapshot**

Similar to the indicator, the modification indicator  shows changes to *set* variables by counting the number of elements added, modified, or removed at the current snapshot.

Design. The modification indicator creates a bar representing the number of *values* that were added ■, modified ■, or removed ■ in the data. The starting point in the dataset labels all values as “added” and otherwise displays changes between snapshots of the program runtime.

Rationale. This representation allows the programmer to understand the impact of transformations on *set* data at a more granular level than whether or not any change has occurred. This approach can be particularly important for understanding the behavior of datasets that frequently filter or update the data during the program runtime.

Stacked Area Chart

LEVEL OF DETAIL: **change**, DATA TYPE: **set**, TEMPORALITY: **sequence**

We selected the stacked area chart  to show the changes within a set variable at the sequence temporality, similar to the modification indicator.

Design. The stacked area chart creates a band representing the number of *values* that were added ■, modified ■, or removed ■ in the data at each snapshot within a sequence of program states, similar to the modification indicator described in the previous section.



Rationale. This representation allows the programmer to see information about the impact of transformations on set data over time. In particular, this behavior can help programmers understand when changes to a set variable are particularly expensive due to unnecessary additions or removals. As with the modification indicator, this approach can be particularly important for understanding the behavior of datasets that filter or update the data values.

7.1.5 Miniaturizations



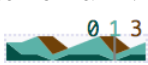
To reduce the **obtrusiveness** of the code augmentations, we designed miniaturizations for the visualizations introduced in the previous section. The miniaturization for the horizon chart and line chart is a smaller version of the horizon chart that appears as an underline of the text. The miniaturization for both the histogram and heatmap is a smaller version of the heatmap that appears as an underline of the text. Finally, for the tick and timeline visu-

alizations, the miniaturization is a compacted version of the full visualization that appears as an underline of the text. We also use the compacted version of the tick visualization as the miniaturization of the stacked area chart. Each of these miniaturizations was designed to provide useful information similar to that of the full visualization but amenable to the size constraints. As the indicator is already small, and the exact value itself is important, we do not provide additional miniaturizations of these visualizations. Examples of these miniaturizations are visible in Figure 7.4, *Expand Inline* and *Expand Below*.

7.1.6 Interaction with Code-Embedded Visualizations

We include a number of interactions to facilitate analysis of the visualizations and enable **comparisons** across representations. For the exact value representation, mousing over a simplified object augmentation `{ "x": 42, ... }` displays the full object `{ "x": 42, "y": 56 }`. For the line chart and horizon chart, mousing over the visualization shows a red cursor and the value at the current point . To facilitate comparison between augmentations, holding shift draws a cursor and value for the current snapshot across all visible charts in gray (as in the next figure). For augmentations representing an object simplification, the cursor displays the property name in addition to the full value .

For the histogram and heatmap, hovering over a bar shows additional details, including the range or value for the current bin and the number of elements. For visualizations representing a simplified object, the augmentation similarly shows the bin and count, but also shows the name of the property that is currently visualized (Figure 7.2). If the programmer holds shift while interacting with the histogram, the environment updates all the related visualizations to show the distribution *relative* to the current selection (Figure 7.3).

Mousing over the tick and timeline visualizations highlights the snapshot and displays the value of the program variable at that snapshot . To facilitate comparisons, holding shift highlights and displays the value for all visible augmentations. For the modification indicator  and stacked area chart , mousing over the visualization shows the number of elements added, modified, or removed from the set variable at the given snapshot.




```

151 {
152   "type": "text",
153   "from": {"data": "drive"},
154   "properties": {
155     "enter": {
156       "x": {"scale": "x", "field": "miles"},
157       "y": {"scale": "y", "field": "gas"},
158       "dx": {"scale": "dx", "field": "side"},
159       "dy": {"scale": "dy", "field": "side"},
160       "fill": {"scale": "c", "signal": "mouseover_year"},
161       "text": {"field": "year"},
162       "align": {"scale": "align", "signal": "[1995, 2002]"},
163       "baseline": {"scale": "base", "field": "side"}
164     }
165   }
166 }

```

Figure 7.3: Code augmentations visualize the runtime state of program variables in Vega [177] code. A *histogram* shows the distribution of variables containing *set* data. Interacting with the *year* histogram filters all other histograms to only show the data values where the *year* is between 1995 and 2002 (hand cursor shown here for clarity; in a real implementation the cursor is hidden mid-interaction to aid chart reading). For *mouseover_year*, a *tick* visualization depicts value changes.

7.1.7 Visualization Color for Code-Embedded Visualizations

We vary the color for the code augmentations based on the *type* of data being displayed in order to facilitate identification of the type when similar visualizations are displayed in the development environment. For each *type* of visualized data, we select a diverging color scheme to encode positive and negative values in the visualizations described in the preceding sections. For augmentations that show object simplifications, we further differentiate the color to attract the programmer’s attention to the simplification (e.g., `index_term` in Figure 7.2). The decision to change the color based on the *type* of the data makes it easier to differentiate between *value*  and *set*  data, as *value* data often represents exact values whereas *set* data performs some aggregation on the underlying data (such as the variability). Further differentiating the color for object simplifications  ensures that these visualizations stand out from both *value* and *set* variable augmentations (Figure 7.2).

7.1.8 Visualization Size for Code-Embedded Visualizations

To support the **comparability** of augmentations throughout the code, we standardize the augmentation width, which is set to a constant value based on the average token size in the programming language. The augmentation height is determined by the line height. The standard size allows programmers to more easily compare between *sequence* augmentations as the visualized time scale is the same [154]. We add additional inter-word and inter-line space as necessary depending on the placement strategy (Section 7.1.9). For certain placement techniques, we use a variable width since the miniaturization and augmentations appear in place. Goffin et al. [66] discuss design considerations for increasing the visualization size to fill the inter-line space or to add additional inter-word padding in more detail.

7.1.9 Placement of Code-Embedded Visualizations

We identified twelve techniques for the placement of code augmentations (Figure 7.4) and assessed each technique based on a set of design trade-offs. These placement options are an extension of the techniques previously presented by Goffin et al. [66].

Placement Techniques

The *right* and *left* placements position augmentations adjacent to the corresponding token. The *above* and *below* placements ensure that individual lines of code maintain their original structure, but may add new whitespace lines that increase the overall code length. The *inline-transparent* and *inline-opaque* placements draw augmentations over the corresponding token, thus requiring interaction to improve legibility of the visualization or code.

To satisfy the need for **unobtrusive** augmentations, we include the placement techniques: *expand-inline* and *expand-below*. These placement techniques require the augmentation to include a miniaturization that displays the augmentation as an underline of the text. Hovering over the token expands the augmentation as either *inline-opaque* or *below*. However, unlike *below*, *expand-below* does not add a whitespace line and instead draws the augmentation over the existing code so as not to require additional spacing in the code.





Placement		Visualization	Reflow	Spacing	Occlusion	Width	Alignment
Right	99 100	"x": {"scale": "x", "field": "date"  , "offset": 2}, "y": {"scale": "y", "field": "indexed_price"  ,	●	○	○	○	○
Left	99 100	"x": {"scale": "x", "field":  "date", "offset": "y": {"scale": "y", "field":  "indexed_price"},	●	○	○	○	○
Above	98 99	"update": { "x": {"scale": "x", "field":  "date", "offset": 2},	○	●	○	○	○
Below	99	"x": {"scale": "x", "field": "date", "offset": 2}, 	○	●	○	○	○
Inline Transparent	99 100	"x": {"scale": "x", "field": "date", "offset": 2}, "y": {"scale": "y", "field":  },	◐	○	●	●	○
Inline Opaque	99 100	"x": {"scale": "x", "field":  "date", "offset": 2}, "y": {"scale": "y", "field": "indexed_price"},	◐	○	●	●	○
Expand Inline	99 100	"x": {"scale": "x", "field": "date", "offset": 2}, "y": {"scale": "y", "field":  },	○	○	●	●	○
Expand Below	99 100	"x": {"scale": "x", "field": "date", "offset": 2}, "y": {"scale": "y", "field": "indexed_price"}, 	○	○	●	●	○
Left Margin	99 100	 "x": {"scale": "x", "field": "date", "offset": 2},  "y": {"scale": "y", "field": "indexed_price"},	○	○	◐	○	●
Right Margin	99 100	"x": {"scale": "x", "field": "date", "offset": 2},  "y": {"scale": "y", "field": "indexed_price"}, 	○	○	◐	○	●
Inline Start	99 100	 "x": {"scale": "x", "field": "date", "offset": 2},  "y": {"scale": "y", "field": "indexed_price"},	◐	○	◐	○	◐
Inline End	99 100	"x": {"scale": "x", "field": "date", "offset": 2}  "y": {"scale": "y", "field": "indexed_price"} 	○	○	○	○	○

Figure 7.4: Twelve placement techniques for code-embedded visualizations shown within a sample Vega [177] specification. Each technique has different trade-offs regarding the amount of inline text reflow, additional line spacing, occlusion of other text, variable width requirements, and alignment of the augmentations within the development environment. A cursor is shown for placement techniques requiring interaction (hand cursor shown here for clarity; in a real implementation the cursor is hidden mid-interaction to aid chart reading). The trade-offs are labeled as required ●, up to the discretion of the implementer ◐, or not required ○.

The *inline-start* and *inline-end* placement techniques position augmentations on the same line, but not adjacent to their corresponding tokens. The *inline-start* placement leverages the code indentation to place augmentations in the whitespace at the start of the line. The *inline-end* placement maintains the full readability of the line, using the augmentations as the final punctuating marks. As multiple tokens may occur on the same line, the augmentations can either be placed adjacently or overlapping, but some interaction is required to relate augmentations back to their corresponding tokens (e.g., via highlighting).

The *left-margin* and *right-margin* placement techniques separate the augmentations from the token by placing them in the margin of the code editor, and in doing so improve **comparability** across augmentations. When multiple augmentations exist on the same line, they overlap, thus requiring additional interaction to select the augmentation of interest.

Placement Trade-off Metrics

For each placement technique described in this section, we discuss trade-offs in the application of the technique with respect to the *reflow* requirements, line *spacing*, augmentation *width*, vertical *alignment*, and *occlusion*. We provide a brief description of each metric below, and include the results of the various placement techniques in Figure 7.4.

reflow The code must reflow the text inline to make space for the augmentation, thus increasing the length of the line.

spacing The code must add additional space between lines to include the augmentation, thus increasing the length of the document.

occlusion The augmentation partially or fully occludes the code, thus requiring interaction to improve legibility of the augmentation or code.

width The augmentation must have a variable (rather than fixed) width to fulfill placement requirements in the code, thus reducing the comparability.

alignment The augmentation will be positioned such that it is vertically aligned with other augmentations in the document to support comparison.

Reflow. The *left* and *right* placement techniques introduce reflow changes inline to make space for the visualization. Depending on the programming language used, these changes may be minimal due to the amount of existing whitespace and structure in the code. The *inline-transparent* and *inline-opaque* placement techniques may require some reflow for small tokens to improve the legibility of the augmentations by increasing their *width*. If many augmentations are on the same line, the *inline-start* placement may increase the indentation at the start of the line. All other techniques do not cause reflow changes.

Spacing. Only the *above* and *below* placements add additional spacing to the document. The impact of this additional spacing is highly dependent on the structure of the code; for augmentations where whitespace is already available above or below the line, we do not introduce a new line but instead use the existing whitespace (Figure 7.4, *Above*).

Occlusion. The *inline-transparent* and *inline-opaque* placements occlude the token, thus requiring interaction to improve the legibility of the augmentation or code. For the *expand-inline* and *expand-below* techniques, the augmentations will not *occlude* the code when miniaturized, but will introduce some occlusion when expanded to their full size. The *left-margin*, *right-margin*, and *inline-start* techniques may introduce occlusion if the augmentations overlap when there are more than one on a given line. For the *inline-end* placement, augmentations can be positioned side-by-side to facilitate comparison. The *above* and *below* placements may need to handle occlusion with other augmentations on the same line, as described by Goffin et al. [66]. Neither the *right* nor *left* placement techniques occludes the text.

Width. The *right*, *left*, *above*, *below*, *inline-start*, *inline-end*, *left-margin*, and *right-margin* techniques use a standard width to facilitate comparisons. For the *inline-transparent*, *inline-opaque*, *expand-inline*, and *expand-below* techniques, the width matches the size of the token.

Alignment. The *left-margin* and *right-margin* augmentations will be aligned, thus facilitating comparisons. Using the *inline-start* placement can produce augmentations that are aligned based on the indentation depth of their corresponding tokens. All other techniques position tokens relative to the original positions and will therefore not be aligned.

7.1.10 General Placement Guidelines

Based on our design considerations, the augmentations must be **comparable**, **unobtrusive**, and **salient**, such that they attract programmers' attention to interesting trends, without detracting from programmers' ability to perform their primary task. These considerations are directly related to the third challenge: "support the tasks that matter most to the user."

The **comparability** of the augmentations is primarily impacted by their *alignment* and *width*. When programmers need to make fine-grained comparisons between augmentations, the *margin-left* and *margin-right* placements are ideal because they maintain the same temporal axis across augmentations and provide a standard location at which to find information.

The **unobtrusiveness** of the augmentations requires that minimal changes be made to the code structure caused by *reflow* or additional *spacing*; large structural changes can be detrimental to programmers' ability to review the code [211]. *Occlusion* is also relevant to the programmer's ability to read the code or extract information from the augmentations at a glance. When the unobtrusiveness of the augmentations is most important, we recommend using the *expand-inline* placement technique; this technique provides an indicator of what variables are updating during program execution while limiting changes to the code visibility.

The **salience** of the augmentations influences how easily the programmer's attention is attracted to particular augmentations of interest. Whereas the *expand-inline* placement provides some indication of variable changes or distributions, it can easily be overlooked as it appears only as an underline of the text. To better attract the programmer's attention, we recommend using the *right* placement technique to produce a large augmentation near the source of the token. For tokens on the periphery of the programmer's attention, the *inline-opaque* technique may be better so as to reduce *reflow* changes to the document.

7.2 Implementation of Code Augmentations for the Online Vega Editor

To explore the utility of these code augmentations in a real programming environment, we implemented a set of embedded visualizations as an extension to the online Vega code editor. In particular, we identified in situ visualizations from our design space that are relevant to novice users and appropriate for the Vega runtime state. We followed a simple rule-based process for selecting the type of in situ visualization to display for each variable in the Vega specification. For this implementation, we do not currently allow the visualization type to change once selected, though such automatic updates are an important part of future work. Figure 7.5 shows an example of the code augmentations in the online Vega editor.

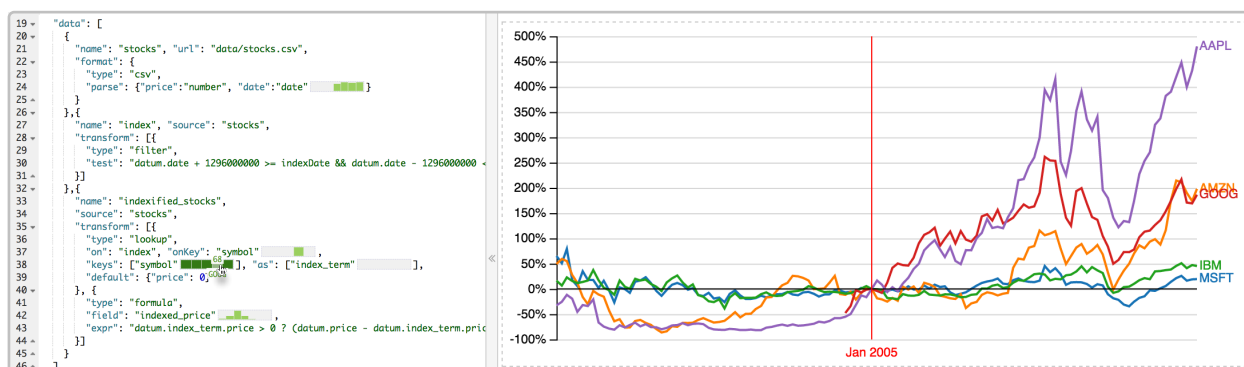
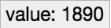










Figure 7.5: Code augmentations visualize the runtime state to show the distribution of data. Interacting with the "symbol" histogram filters all other histograms to only show the data values where the "symbol" is "GOOG" (Line 38, hand cursor shown for clarity; in the editor the cursor is hidden mid-interaction to aid chart reading). This interaction shows that "GOOG" has missing data for early dates in the dataset (Line 24), which results from a later IPO date than the other companies.



Tooltip  Hovering over a data field or signal name shows the current state of the variable on-demand, similar to existing debugging environments [9]. For data fields, the tooltip shows summary information about the dataset (e.g., the min, max, and mean value for that field). For signals, the tooltip shows the current value. See Appendix E.3 for an example figure showing the tooltip behavior (Figure E.2).

Indicator  For signal variables that are not expected to change (e.g., do not include an “update” clause or react to input events), we selected the indicator to deemphasize the signal definition. The indicator also allows for quick value extraction.

Line  For signal variables where the type at runtime is a number, we select the line visualization because it represents the range and value of the variable over time, rather than a snapshot of the change. For signal variables that are an array of numbers, we use a different line for each element in the array  to fully represent the variable.

Tick  For signal variables that are initialized to null or are not a number at runtime, we selected the tick visualization to emphasize when changes occur and to show the behavior of the variable over time. We selected the tick visualization rather than the timeline  because the tick visualization explicitly represents each state in the program runtime and may thus be more informative for novice users.

Histogram  We selected the histogram for data field variables because it visualizes the current state. While the horizon chart  can highlight substantial shifts in the historical values of the variables, this visualization summarizes the variable rather than explicitly representing the underlying values. The horizon chart is also more likely to be unfamiliar to novice users and may thus require additional training to interpret. The histograms can highlight changes between states by observing shifts in the distribution. Since position encodings are more interpretable than color encodings [132], we chose not to utilize the heatmap  visualization for our final implementation.

We selected this set of visualizations to provide ones that were likely to be familiar to novice Vega users and thus interpretable with minimal training. While this selection only covers a subset of augmentations from our design space, we did first prototype all the augmentations proposed in this system, as well as the various placement techniques. Other visualization designs not included in this implementation may be useful for specialized tasks by expert users. For example, the modification indicator  or stacked area chart  could be useful for visualizing tuple-specific changes to the underlying Vega datasets. These visualizations could therefore be particularly useful for debugging tasks surrounding data transformations. Future work should further explore techniques to enable users to customize which visualizations are displayed for different program variables.

7.3 Evaluation: Understanding Program Behavior of Vega

To evaluate the utility of in situ visualizations for program understanding tasks, we conducted a user study with 18 novice programmers. Participants were presented with two unfamiliar Vega programs and asked to answer 18 program understanding questions about the behavior (see Appendix E.6). We compared our in situ visualizations implemented in the online Vega editor to a baseline condition in which participants were only able to use a tooltip to inspect individual values. Participants completed a post-task questionnaire (Appendix E.7) in which they rated their self-perceived speed and accuracy on the task questions on a scale

from 1 (better with the baseline condition) to 7 (better with the visualization condition). Participants also scored how helpful, interpretable, and intrusive each of the in situ visualizations were on a scale from 1 (not) to 5 (extremely). For the evaluation, we selected the *right* placement to ensure that the in situ visualizations are salient and in close proximity to their corresponding token. This placement choice was motivated by Zellweger et al.’s [211] discussion of the importance of proximity for embedding contextual information. Furthermore, the *right* placement reduces the overall reflow of the document and follows the reading direction of the code. Appendix E includes the various resources used for the evaluation.

Participants. We recruited 18 participants (11 male, 7 female) from the University of Washington. Participants included both PhD (4) and undergraduate students (14). Participant ages ranged from 18 to 30 (mean 21.1, s.d. 3.74). Participants completed a screening survey about their experience and programming language familiarity to ensure participants had prior programming coursework or job experience (Appendix E.1). The most common programming language regularly used by our participants is Java, followed by Python, JavaScript, and C/C++. All of our participants were novice Vega users (i.e., were unfamiliar programming in Vega), though two participants had previously seen Vega in other contexts. Each participant received a \$20 gift card for completing a 90 minute session.

Methods. Participants answered program understanding questions about two Vega programs, with and without the assistance of in situ visualizations of the program behavior. At the start of the evaluation, participants were given an instruction sheet with a sample Vega program, an explanation of the code, and an introduction to the development environment and important keywords (Appendix E.3). Participants were then given a training task in which they answered several sample questions and viewed the sample answers (Appendix E.5). During this time, participants were encouraged to ask any questions about Vega or the task setup. Once participants started the tasks, we no longer answered questions. For the study tasks, we selected four Vega programs that cover a range of visualization designs, datasets, and program understanding challenges. Three of these scenarios exhibit an error in the behavior.

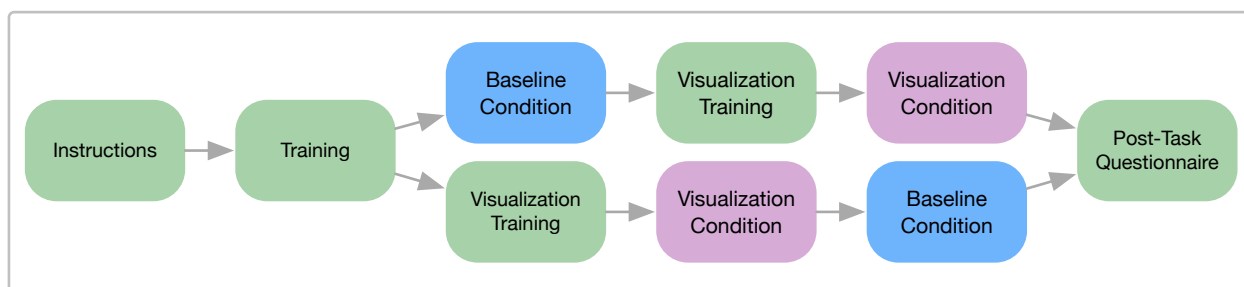


Figure 7.6: Participants completed two evaluation tasks, one in each of two conditions (*baseline* or *visualization*); we counterbalanced the conditions across participants. This figure shows the steps followed in the evaluation, with each participant completing only one path.

Population. A population pyramid includes a slider to select the year that is visualized. This Vega specification includes missing data for the year 1890, which causes derived datasets to be empty and the visualization to be blank at this point.

Index. A line chart of stock prices with an interactive cursor that renormalizes the data relative to this point to show the return on investment. The Vega specification includes derived datasets and nested data declarations. An error in one of the data transformations causes all tuples to be filtered out at certain points, causing the lines to visually flatline. This example was previously described in Section 6.2.1 and Appendix B.

Scatterplot. A scatterplot of points that supports infinite panning. This Vega specification includes many interconnected signal definitions to define the interaction. Due to a bug with the evaluation order of signals, the domain of the axes becomes distorted while panning. This example is similar to the one described in Section 6.2.2 and Section 7.1.

Overview. Two area charts showing stock price over time; selecting a region in the smaller chart zooms the larger one to produce a filter+context visualization. This Vega specification includes many interconnected signal definitions nested in the specification. There is no error.

Participants completed two tasks, one in each of two conditions: *baseline* and *visualization*. In the baseline condition, participants were given a simple code editor based on the online Vega editor. Tooltips were added to the signal and data field tokens to show information about the runtime state, similar to other common development environments [9]. For the visualization

condition, the editor was additionally augmented with our in situ visualizations. Prior to the visualization condition, participants were shown an instruction sheet with an explanation of the in situ visualizations (Appendix E.4) and were encouraged to experiment with the augmentations using the same visualization as the training task. We counterbalanced the order of the conditions across participants. The experimental protocol is shown in Figure 7.6.

Participants answered 18 program understanding questions for each task about major Vega concepts, such as *signals*, *datasets*, and *data fields*, which required participants to reason about how the visualization state changes during interaction. These questions encouraged participants to read and experiment with the program to develop an understanding of the interconnectedness and runtime behavior of the code. Participants were asked to identify any unexpected behavior in the Vega output and answer follow-up questions about the source of that unexpected behavior; participants were not informed that an error existed if they did not identify it themselves. Participants provided free-form answers to each question and rated their confidence on a scale from 1 (not confident) to 5 (extremely confident). The list of program understanding questions used for the evaluation is included in Appendix E.6.

At the end of the session, participants completed a post-task questionnaire (Appendix E.7) in which they rated their self-perceived speed and accuracy on the task questions on a scale from 1 (better with the baseline condition) to 7 (better with the visualization condition). Participants also scored how helpful, interpretable, and intrusive each of the in situ visualizations were on a scale from 1 (not) to 5 (extremely). The larger scale for the speed and accuracy was selected to encourage nuanced comparison across the conditions.

7.3.1 Quantitative Results & Analysis

To perform the analysis, we first created a gold-standard set of answers for each task and scored participant answers on an integer scale from 0 to 2 (“incorrect,” “partially correct,” “correct”). Scores on each participant’s answers were provided by the second author for this work (Arvind Satyanarayan), who was blinded to the study condition for each task. Final grades for each participant were determined by simple summation.

We fit linear mixed-effects models for participants' grades, log-transformed task times, and average confidence. Each model included fixed effects for condition and presentation order, plus per-subject random intercepts. Likelihood ratio tests indicated a marginally significant effect of the visualization condition on task grade ($\chi^2(1) = 3.30, p < 0.1$). Participants had roughly one more "correct" (or two more "partially correct") answers in the visualization condition overall. On average, participants had a score of 22.5 in the baseline condition and about 24.5 in the visualization condition (out of 42 total points). Exploratory data analysis indicated a strong difference in grades due to education level, but with similar absolute grade improvements in the visualization condition. There was a significant effect of task order on the log time for participants to complete the task ($\chi^2(1) = 8.96, p < 0.01$), with participants faster in the second task regardless of condition. We found no significant effect of condition or order on participant confidence. For the post-task questionnaire, we used 1-sample nonparametric Wilcoxon signed rank tests with a null hypothesis that the result is neutral (middle Likert scale value). We found significant positive effects in favor of the in situ visualizations for participants' self-reported speed ($p = 0.002$) and accuracy ($p = 0.0026$).

Figure 7.7 depicts subject ratings of how helpful, interpretable, and intrusive each of the visualizations were. We found a significant positive effect for line visualization helpfulness ($p = 0.005$) and a marginally positive effect for histogram helpfulness ($p = 0.186$). We found a significant positive effect for how interpretable the value ($p = 0.003$), indicator ($p = 0.021$), and line ($p = 0.015$) visualizations were, and a marginally significant positive effect for the histogram ($p = 0.132$). For the visualization intrusiveness, we found a significant negative effect for the value ($p = 0$), indicator ($p = 0.005$), line ($p = 0.009$), and tick ($p = 0.031$).

7.3.2 Qualitative Results & Discussion

We selected questions that would be reasonable to expect participants to answer regardless of condition. The notable significant effect of task order on the completion time suggests some improved knowledge of the language and questions, which helped participants know where to look. While we did see a marginally significant effect of the visualization condition on task

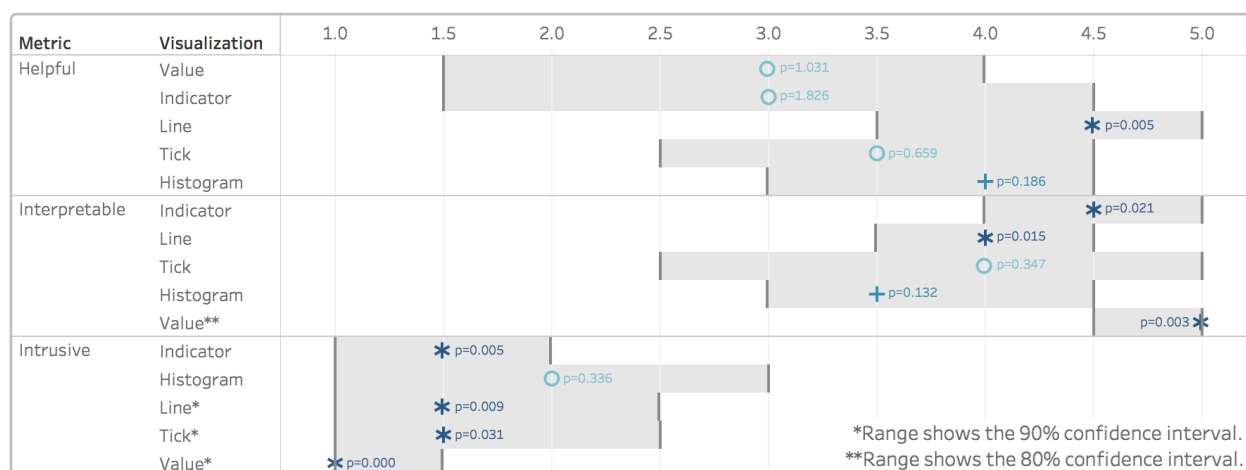


Figure 7.7: The pseudo-median value and 95% confidence interval (unless otherwise noted) for how helpful, interpretable, and intrusive each code augmentation was on a scale from 1 (not) to 5 (extremely). Median values are labeled with the p-value for the 1-sample Wilcoxon signed rank test. Note: the Wilcoxon rank test could not compute the full 95% confidence interval for scores tightly clustered near one or five, and therefore show the confidence interval otherwise noted in the figure: *For the intrusiveness of the line, tick, and value, the range shows the 90% confidence interval; **For the interpretability of the value, the range shows the 80% confidence interval.

grade, participants were generally able to answer the task questions by reviewing the code and probing the state information with the tooltip on signals and data fields. In particular, there were a number of question and task combinations for which all participants provided the correct answer, regardless of condition. For example, the first question in the task asks: “What is the name of the primary dataset being visualized?” 34 out of 36 answers on this question were correct; the two incorrect answers occurred during the baseline condition.


The question answering process could be quite different between the two conditions. Question 8 asked participants to identify how each signal that updates is used throughout the code. In the baseline condition, P18 spent over 16 minutes attempting to identify how each signal in the specification behaved (Q8 median 3.65 min). To fully answer this question, P18 carefully experimented with different interactions, probing the signal value with the tooltip to identify when it changed. This back and forth between testing interactions and assessing the state clearly demonstrates the disconnect between the code behavior and output. Participants in the visualization condition similarly tested interactions, but could identify changes at a glance. As P2 put it: “the [*in situ*] visualizations allowed me to connect the

dots between the code, its properties, and what it did.” When comparing the two conditions, P11 noted that *“The biggest factor for me was just seeing which values change in real-time when interacting with the visualization.”* Rather than reading the code or probing the state on-demand, participants could view changes as they worked rather than as a separate task.

We saw a significant positive effect in favor of the in situ visualizations on both participants’ self-reported speed and accuracy. Across conditions, participants utilized search to find keywords of interest. But, as P15 explains, *“the code visualizations helped better locate the signals and made me more confident about my answers.”* Moreover, the in situ visualizations turned the underlying data into a physical artifact to reason over. P2 noted that *“I found it helpful to be able to interact with the data on a graphical, physical level.”* Unlike our prior work on visual debugging techniques, the in situ visualizations were more concretely grounded in the source code. Instead of requiring participants to shift their attention amongst different views, the embedded code visualizations reduced the separation to allow participants to focus on the program understanding process within the code context.

The error in the index chart is a particularly challenging one to reason about because it occurs due to a small difference in the date/time of the tuples in the dataset compared to the filter window (see Appendix B). For most participants, it was clear that the error only occurred at certain dates in the visualization. However, the error was not with the signal itself. For P9, it became clear that the error was in the data because *“when I mouse over the flatline behavior, the ‘index’ field changes and the [histogram] shifts to the left (next to the variable).”* When interacting with the in situ visualization of the signal, P9 also correctly noticed that the `indexDate` contained a time in addition to the date. While none of the participants correctly diagnosed the error in its entirety, P9’s observations were crucial steps towards uncovering the convoluted source of this error. The in situ visualizations provide a lightweight way to incorporate the underlying state into the program understanding process.

While the utility of in situ visualizations for interpreting interactive behavior was apparent, participants sometimes struggled to understand the contextual implications, particularly for the data fields. For instance, to understand the range of data fields in the index chart, P9

inspected one of the visual encodings and noted “*Oh, okay, I guess Microsoft... for some reason.*” Although P9 identified that the data field only had one value (which was Microsoft), it was not clear why this was the case when other parts of the code showed the full range (e.g., all five companies ). In this case, P9 had overlooked the nested dataset declaration; five different marks are dynamically created, but the embedded visualizations only show the results for one of them. To understand where each data field came from, participants needed a more intimate understanding of the implementation hierarchy and data flow.


7.4 Limitations and Future Work

While Vega’s relative simplicity was convenient for evaluating the design space of in situ visualizations, Vega is an exemplar of a larger class of reactive languages, such as React [93] and Elm [36], for which these techniques could apply. For example, Elm [36] was originally designed as a reactive programming language that similarly utilizes the streaming constructs implemented (and now visualized) in Vega. Moreover, many of Vega’s constructs are representative of properties seen in other languages. The arbitrary nesting of mark definitions produces numerous examples of how scope can be a concern when referencing particular variables and how such environments are dynamically allocated at runtime. Since marks can be dynamically added and removed, the scope and number of instances can change as the programmer interacts with the output Vega visualization. While the design space presented in this chapter provides a breakdown of potential data types of interest (Figure 7.1), we do not currently address this underlying code structure. The user evaluation (Section 7.3) surfaced some of the challenges in this space; as discussed in Section 7.3.2, P9 struggled to interpret the nested dataset declaration when viewing derived code augmentations for the line marks in an interactive index chart. The use of object simplification and summarization in the embedded visualization shows one approach to handling complex nested data structures by emphasizing particular properties of interest. However, future work should explore how interaction or other techniques might surface relevant scoping information or better help users to navigate complex nested data structures in the code.

Vega’s reactive framework was useful for effectively snapshotting the program behavior to produce the in situ visualizations. Section 5.2.1 provides some additional discussion about the design of Vega as a framework on which to explore new program understanding techniques. While we do not propose advances in system logging, techniques such as Dolos from Burg et al. [26] enable efficient logging with minimal overhead for web programming. This infrastructure could provide an effective framework on which to introduce in situ visualizations for web development contexts. However, such an extension also raises new questions about how best to identify relevant code variables for imperative programming domains. While techniques such as static analysis or manually added annotations for important variables [36] can help address this problem, future work should explore new techniques that can preemptively surface interesting details before the programmer becomes aware that a problem exists, and adapt to changes in the programmer’s workflow.

The goal of these in situ visualizations was to enable the system to better “support the tasks that matter most to the user” by communicating relevant details of the program behavior directly within the code. By utilizing this approach, programmers can view important information while otherwise focused on authoring or reviewing code. While in situ visualizations can helpfully call attention to the dynamic behavior of program variables, it may be important to change their salience relative to the programmer’s current task. P14 noted that *“I thought the code visualizations were cool and helpful, but they could also be a little distracting.”* In order to appropriately incorporate inline visualizations for real-world use cases, future work should explore how the style and placement of inline visualizations could dynamically adapt based on the programmer’s current task to only employ in situ visualizations as needed. While the work presented in this dissertation moves one step close towards recognizing and supporting the user’s primary task, future work is required to identify how best to adapt program understanding approaches to the ever-changing needs of the user.

7.5 Summary of Contributions

Vega allows programmers to focus on authoring interactive visualizations while deferring implementation details to the underlying Vega system. However, this approach to visualization design introduces a gap between the code written by the programmer and the system output, which complicates the program understanding process particularly when debugging the time-varying behavior of end-user interactions. To alleviate the need to switch amongst different program understanding tools (see Chapter 6), we contribute a design space of in situ visualizations  that appear directly inline in the source code to support program understanding during all phases of the development process, not just while debugging.

In the formative interviews on new debugging techniques for Vega (Section 5.4) one of the participants stated that Vega “*need[s] a way to examine internal variables... [and] to see the internals of the step-by-step process.*” In an evaluation of this work (Chapter 6), we showed that new visual debugging techniques can in fact help programmers accurately trace changes through the code to identify bugs or crucial dependencies. While the visual debugging techniques helped with program understanding, there still remained a disconnect between the user-authored code and visual debugging techniques. The in situ visualizations presented in this chapter further reduce this gap. In evaluating these results (Section 7.3), one participant explained that “*the [in situ] visualizations allowed me to connect the dots between the code, its properties, and what it did.*” By leveraging the domain-specific concepts that are familiar to the user (e.g., the program variables and interaction logic), we were able to design visual debugging techniques that address the debugging needs surfaced in our early formative interviews. As with the original visual debugging techniques, these visualizations reflect the level of abstraction with which users are most familiar, but were specifically designed to better support users in completing the tasks that matter most—such as authoring or reviewing code—by communicating relevant program details directly within the code.

This work was done in collaboration with Arvind Satyanarayan and Jeffrey Heer, and was originally published and presented at CHI 2018 [88].

Chapter 8

AUTHORING AND REUSING RESPONSIVE VISUALIZATION DESIGNS

Visualization plays an essential role in both performing data analysis and communicating analysis insights. The previous chapters have explored new strategies to produce graph visualizations that encode domain knowledge (Chapter 4) and new techniques to support program understanding for interactive visualization design (Chapter 5-7). This chapter explores the final piece of this process: customizing and annotating visualizations for communicative purposes. A crucial component of this step is the design of responsive visualizations. Responsive visualizations adapt the visualization content based on the screen size or interactive capabilities of the viewer's device (e.g., a mobile phone or tablet).

Consider the use of visualizations in news articles. Mobile devices are now a more important platform than desktop or laptop computers for consuming news articles [200]. While the text content of the article may easily adapt to the device size, it is non-trivial to create responsive visualizations. Responsive visualizations must adapt the design so that content remains informative and legible across different device contexts. For example, designers may choose to resize certain visualization marks or swap the axis encodings so that a chart fits better on a mobile screen. Designers may also employ different interaction techniques depending on the interactive capabilities of the device. Despite the necessity of responsive visualizations, the process of developing and maintaining multiple designs requires extensive user time and effort. Responsive visualizations therefore become a burden on the designer's development workflow. While responsive considerations may be discussed in the abstract throughout the visualization design process, implementation of the responsive design often occurs only in the final stages once the idea for the visualization has been fully formed.

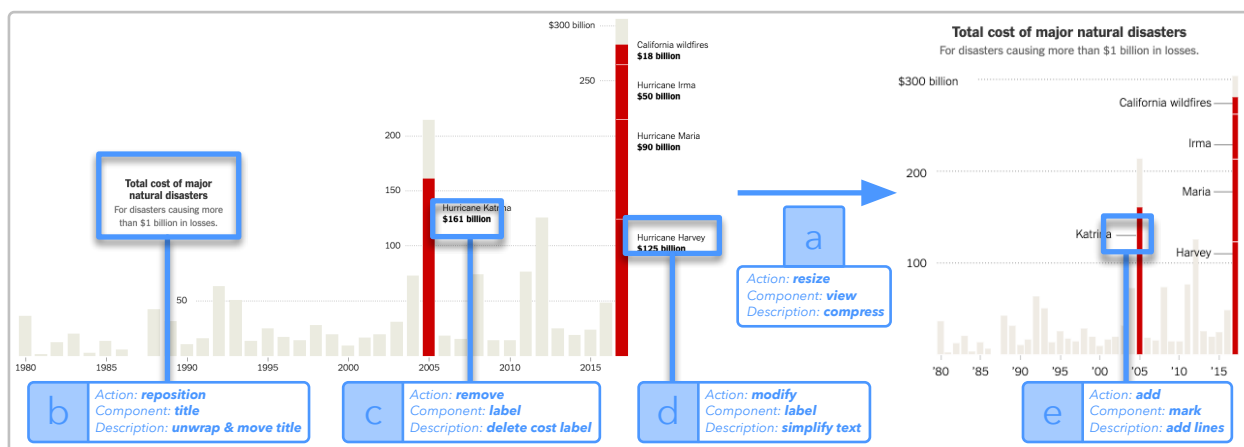


Figure 8.1: Desktop (left) and mobile (right) visualizations from the New York Times article “The Places in the U.S. Where Disaster Strikes Again and Again” [G13]. This example demonstrates responsive techniques that: (a) *resize* the view to compress the width; (b) *reposition* content (e.g., axes, labels, and title); (c) *remove* unnecessary labels; (d) *modify* the text and axis labels to reduce complexity; and (e) *add* new line marks to connect labels to the corresponding bars.

To understand current practices for responsive visualization design, this chapter first contributes a survey of 53 news articles gathered from 12 sources. From these articles, we identified 231 visualizations and labeled the visualization type and responsive techniques used by the article. Figure 8.1 shows an example visualization from this corpus as well as several of the responsive techniques used to modify the design. Based on this survey, we identified six high-level actions performed on individual components of a responsive visualization design: *no change*, *resize*, *reposition*, *add*, *modify*, and *remove*. The most common action in our corpus is to remove content from the mobile visualization. However, visualizations often exhibit multiple techniques, including more complex customizations such as completely redesigning the visualization encoding or adding clarifying marks. While a few designs take into account device-specific interaction capabilities, the vast majority of adaptations focus on creating legible charts at different sizes corresponding to different device categories (e.g., desktop, tablet, or portrait orientation on mobile). Thus, in this chapter we use the terms “device size/context” and “chart size” interchangeably to describe the responsive visualizations.

This chapter further contributes interviews with five authors of the coded visualizations to better understand their development processes and the responsive visualization techniques used in their work. These interviews surface some of the core challenges faced by journalists

for producing responsive visualization design. In particular, we find that responsive designs regularly require authors to maintain different artboards for different device sizes. This difficulty demonstrates the complexity of the third challenge explored in this dissertation: “support the tasks that matter most to the user.” While responsive visualization design is an essential component for developing communicative visualizations for news contexts, existing tools rarely provide the support needed to handle this task. Journalists are therefore required to adapt their own preferred workflow around the capabilities of the tools at hand.

To better address these development needs, we identify four central design guidelines to inform the development of new systems for responsive visualization design: (1) **enable simultaneous cross-device edits** to facilitate design exploration for multiple target devices; (2) **facilitate device-specific customization** to address the need for adaptive designs; (3) **show cross-device previews** to provide an overview of customizations applied across devices; and (4) **support propagation of edits** to reduce user effort and accelerate design iterations. Based on these guidelines, we contribute a set of core system features that allow designers to view, create, and modify multiple device-dependent visualizations. These features raise the level of abstraction from a single visualization specification (as in the previous chapters) to generalize the development process for multiple linked views. To this end, our system displays separate views for each chart size and supports simultaneous editing across views. The system then enables generalized selections and view control to support robust customization of marks. Finally, the system foregrounds the variation between visualizations to help designers assess the full picture of the applied modifications and propagate changes.

To demonstrate the utility of our system, this chapter contributes four reproductions of real-world examples from our corpus [G13, G36, G50, G52]. These examples represent a range of visualization types (bar chart, dot plot, line chart, and symbol map) and exhibit unique workflows that demonstrate our core system features. For each example, we provide a step-by-step walkthrough of the development process for the visualization design. These walkthroughs demonstrate how a designer can construct, compare, customize, and iterate on different visualizations using a flexible development workflow for responsive visualizations.

8.1 *Related Work: Responsive Web Design and Mobile Visualization*

While responsive visualization design is still a nascent area, responsive web design has received more attention. Patterns and principles of responsive web design have been extensively studied [133, 134]. HTML5 and CSS3 are popular standards to implement responsive designs [55]. Techniques for responsive web design, however, are not directly transferable to visualization: webpages primarily employ text wrapping, image resizing, and document reflow to achieve responsiveness; these approaches offer little insight on visualization challenges such as data encoding, scale adjustment, or annotation placement.

Responsive visualization becomes particularly necessary for a journalism context in which readers often consume content on mobile devices. Conlen et al. [34] describe techniques to examine reading behaviors for interactive articles, with an implementation targeting Idyll [33]; the articles analyzed by Conlen et al. [34] were primarily designed for desktop use, but 30%-50% of readers consumed and interacted with the content on a mobile device despite the limitations, which suggests that *“mobile users are willing to engage with interactive content, and that the specific interactions should have been refined to better accommodate them”* [34].

Despite the need for responsive articles, there is limited support for designing responsive visualizations. Journalists often combine a variety of approaches including data analysis in R and Python, dynamic visualization development using D3.js [23], and customization in Adobe Illustrator. Static visualization approaches require designers to implement and maintain multiple artboards, which can be time consuming and labor intensive. The New York Times developed ai2html [190], which converts Adobe Illustrator documents into a web format by separating the text and graphic components; this approach ensures that the visualization text remains legible by supporting dynamic placement and scaling, but does not explicitly promote considerations for mobile visualization [168].

Datawrapper [64] is a tool created specifically for journalists to design interactive and responsive visualizations based on a set of templates and device sizes. Datawrapper makes it easy to preview the design across devices, but limits the customization options available

for the visualization designs and narrative content. Power BI has also introduced an automatic approach to responsive visualization design for mobile dashboards [59]. Recent work discusses the application of responsive web design techniques for responsive visualization [1] and strategies for designing visualizations for both desktop and mobile devices [24].

There are several more general approaches for resizing and utilizing the device context for visualization construction. D3.js [23] and Vega-Lite [176] enable the construction of dynamic visualizations that can automatically resize to the available space. Charticulator [161] enables automatic chart layout using constraints and can constrain the layout to fit within a particular artboard size. ViSizer [207] is a framework for applying local optimizations to more effectively resize a visualization. Vistribute [90] is a system for assigning interactive visualizations amongst multiple devices based on properties of the visualization and device.

Visualizations for news contexts also require extra communicative elements and interactive considerations. Idyll [33] is a language for authoring interactive web articles, including the design and parameterization of visualizations. ChartAccent [159] enables free-form and data driven annotations of visualizations. Ellipsis [173] is a tool for authoring visualizations without programming by describing the narrative structure through distinct scenes.

8.2 Formative Interviews: Responsive Visualization Design Practices

To better understand existing responsive design practices, we conducted semi-structured interviews with five journalists about their development approach and rationale for the responsive visualization techniques they employ.

Participants. We recruited five journalists selected from authors of items in the responsive visualization corpus described in Section 8.3.1. All participants had previously published at least one article that exhibited responsive visualization techniques and were personally responsible for the visualization design in the article. We omit demographic information from this section to protect the anonymity of the interview participants. Each interview lasted about one hour; participants did not receive compensation.

Data Collection. The interviews took place over the phone. We captured audio recordings for later review and transcribed notes during each interview.

Protocol. Participants were asked to describe their general process when developing a visualization for a news article and the responsive techniques used in one (or more) of their published articles. Over the course of all interviews, we discussed ten articles from five news organizations. Finally, participants were asked to describe the major challenges they face when designing responsive visualizations. We used a common, semi-structured interview question template across all interviews (see Appendix F).

Analysis. After the interviews, we reviewed the recordings and transcripts to extract common themes regarding the challenges and approaches used for responsive visualization design. These themes are discussed in more detail in the following sections.

8.2.1 *Desktop-First or Mobile-First Development*

Our participants generally described a desktop-first development approach for designing responsive visualizations. Part of the rationale for desktop-first development was that *“by virtue of sort of sketching graphics on my laptop or on my desktop screen, often the first iteration of something works best at those screen widths”* (P3). Another participant explained that *“It’s easier to try things and to come up with an idea... on desktop, cause that’s where we work”* (P2). For the visualizations, one participant noted that *“I think it’s easier to sort of be ambitious when you have a larger palette”* (P1). Designers were generally motivated by the flexibility and ease provided by a desktop development environment, such that mobile designs were not at the forefront of their minds.

While our participants noted that desktop development was often their primary focus, participants also mentioned that they kept the mobile version of the visualization in mind throughout the development process. One participant explained that *“when we’re sketching something and deciding whether something is gonna work, the question of... how is it gonna work on a phone comes up before we’ve gone too far”* (P3). Another participant noted that *“I guess it is always in the back of our minds, like ‘how will this work on mobile’ and often*

we will use that as a rationale to simplify ideas early on in the process because we know that they won't really work on mobile" (P1). While designers may think about the mobile version, they are not necessarily exploring the mobile designs in a practical sense.

Some participants did explain that mobile-first development could be advantageous by encouraging more careful design and simplification of the content. In particular, mobile-first development can help designers *"focus on what's essential"* (P2) and *"it makes us more concise and it makes us get to the point quicker"* (P4). When reflecting on the trade-offs of mobile-first or desktop-first development, one participant noted that the focus was *"Aspirationally, certainly mobile phones. I think in practice, that doesn't really happen"* (P3). Another participant observed that *"much of the programs we use are geared towards desktop first or feel that way, anyway, so if all of them had a slight shift in default or in tone I feel like that would also help us to think that way"* (P4).

8.2.2 Adapting Desktop Visualizations for Mobile

When producing responsive visualizations, participants noted that they would finalize the desktop design before creating the mobile version. As one participant explained, *"the mobile version comes after, when I'm happy with the desktop version, to avoid too many changes"* (P2). To adapt the visualization to a mobile context, our participants often mentioned the need to prioritize information and remove unimportant content. For one example, the participant explained that *"I do remember now removing all of the annotations from that map and I think that was because those annotations weren't fundamental"* (P1). Another participant explained that *"There's a hierarchy of information, right? So as you go down in the artboard size you make the decision about what information can be cut first"* (P3). When reflecting on the adaptation process, another participant explained that *"I think it's easier to eliminate things when you have everything"* (P2). For many of our participants, the most common workflow was to start with the full desktop visualization and to select what content could be removed when scaling visualizations down to mobile sizes; this trend also matches the overall preference for removing rather than adding content, as described in Section 8.3.1.

8.2.3 Artboards, Dynamic Designs, and Automatic Techniques

To produce responsive visualizations, many of our participants chose to focus on a set of pre-defined artboard sizes. However, a major challenge with multiple artboards is maintaining and propagating changes to the design. One participant noted that *“It’s annoying when you have to make changes to three or four or five different artboards and that usually introduces mistakes... so that’s one of the reasons why the design for mobile comes later”* (P2). To produce multiple designs can be a time consuming and labor intensive process. One participant explained that it is *“not the most intellectually stimulating exercise to redesign or make your graphic work... but it is something that needs to be done for every single graphic”* (P3). Another participant noted that *“it feels like a chore... You want to be working on the story; you want to not be working on polishing things for small audiences”* (P1). While there are clear benefits to responsive visualization design, the process of producing these alternative designs can feel like a hindrance to the overall development workflow.

Several participants discussed the use of D3 [23] for easily producing responsive visualizations. One participant mentioned using D3 for a design and the need to dynamically resize the window to test the responsiveness: *“We more just change the width of the screen pixel by pixel to make sure every pixel is properly looking okay”* (P5). However, one participant expressed a hesitance towards dynamic artboard resizing because *“dynamically positioning things like labels and annotations at every possible screen width is very easy for that to go wrong and having a fixed number of breakpoints tends to be a little bit less error-prone”* (P3). While the ability to make designs dynamic could be helpful for producing visualizations that work for any screen size, testing the full range of possibilities was a common source of difficulty and undesirable user effort during the development process.

While there are a variety of common approaches for responsive visualization design (e.g., removing content or simplifying labels), there is not necessarily a straightforward procedure for identifying when and how to employ these techniques. When reflecting on the general workflow, one participant noted that *“I think that it’s usually a pretty iterative, ad hoc pro-*

cess. It takes a bit of thinking. It's usually not the same solution for any two graphics" (P3). Participants often noted that responsive designs were an essential component to their work, but that the development process was currently underserved by existing tools. This observation reinforces the challenge that systems must “support the tasks that matter most to the user.” While many visualization constructions exist for different visualization use cases (see Section 2.1), these approaches do not currently address responsive visualization design.

8.2.4 Takeaways from the Formative Interviews

Participants mentioned benefits of both a mobile-first and desktop-first design approach. Mobile-first encourages designers to focus on only the most important aspects of the data whereas desktop-first development enables more complex, creative, or impressive designs. Since development often happens on a desktop, the designs tend to reflect this default rather than the personal development preferences expressed by our participants. Participants felt that designing for multiple screen sizes (especially mobile) early in the process can lead to better and more consistent cross-device design decisions. More specifically, working through the challenges of visualizing data for various devices helps designers decide what information is most critical, how to effectively highlight key characteristics, and how to effectively encode or layout the data. Empirical evidence suggests that working on multiple prototypes in parallel leads to better and more diverse designs, and increased self-efficacy [39].

However, cross-device design with existing tools is tedious and error-prone because each visualization is treated as a separate artifact, which requires edits to be manually duplicated across designs. While having direct control is important for ensuring that designs meet publication standards, too much repetition discourages iterative design modifications. As a result, most workflows start with a fully-executed desktop design that is modified to better fit mobile screen sizes. While expedient, this approach limits the amount of cross-device design exploration and can lead to inconsistencies between the designs for various devices. This discussion reinforces the challenge that new systems should “support the tasks that matter most to the user,” rather than requiring users to adapt to particular system defaults.

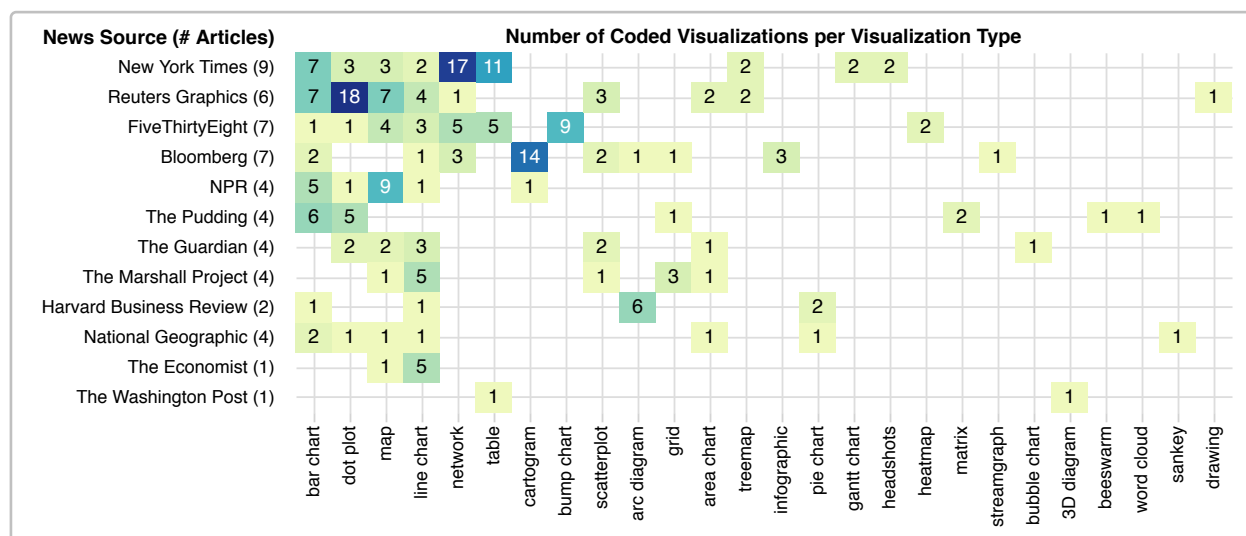


Figure 8.2: We examined 231 visualizations from twelve sources to inform our analysis of responsive visualization techniques. The number of articles per source is shown in parentheses. For the full list of articles analyzed in this work, see Appendix G. We labeled each visualization with the core visualization type. However, some visualizations were more complex (e.g., a normalized, stacked bar chart); 46 of the 231 visualizations were small multiples.

8.3 Techniques for Flexible Responsive Visualization Design

To better understand the space of responsive visualization techniques currently employed in news articles, we first analyzed a corpus of responsive visualizations from 53 news articles (Section 8.3.1). Based on this analysis and the results of the formative interviews described in Section 8.2, we identified a set of four design goals for new systems to promote responsive visualization design (Section 8.3.2). We then propose a set of core system features to better support flexible responsive visualization design (Section 8.3.3).

8.3.1 Responsive Visualization Corpus

We selected 53 news articles gathered from twelve sources—including the New York Times, Reuters Graphics, and FiveThirtyEight—to produce a corpus of 231 responsive visualization examples. To build this corpus, we surveyed best-of lists and selected articles that included at least one visualization exhibiting responsive techniques. We made sure to select articles that include a wide range of different visualization types. We labeled each visualization instance with the visualization type (Figure 8.2) and responsive techniques used when adapting the

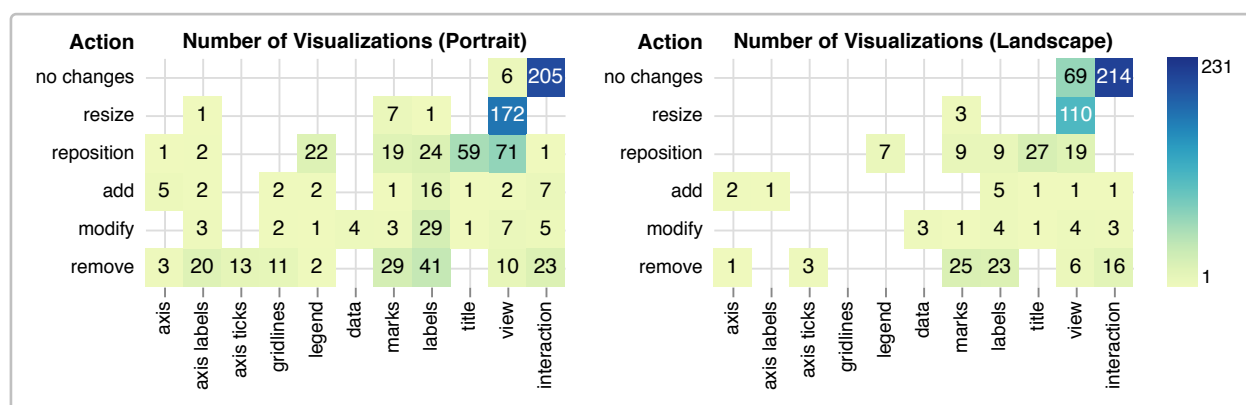


Figure 8.3: We performed an open coding of the responsive techniques used for the portrait (left) and landscape (right) orientation of a phone. The labeled techniques reflect the changes made from a desktop visualization to the mobile visualization. We then clustered the techniques to indicate the type of action and the component to which it applies. Responsive techniques were used much more frequently to customize the portrait visualizations than the landscape visualizations. It was also more common to remove or reposition content, than to add new content for the mobile version.

design between the desktop and mobile versions of the visualization (Figure 8.3). For the analysis, we performed an open-coding of the responsive techniques for the visualization design and interactive techniques used. Two of the authors coded and discussed a set of overlapping visualizations to ensure inter-coder agreement. When labeling the responsive techniques, we identified changes from the desktop to the mobile version of the visualization. To view the mobile version of the visualization, we used the Device Mode [10] provided by Chrome DevTools to simulate an iPhone X device. We then examined the responsive techniques used for both the portrait and landscape orientation of a phone. Figure 8.1 shows several of the open-coding labels generated for the visualization (e.g., the description); we provide the full list of open-coding labels generated for seven visualization designs in Appendix H.

We grouped the codes based on their behavioral similarity to determine the core *editing action*, and we associated the action with a particular *visualization component*. The responsive techniques generally fall along a spectrum of simple editing actions: *no changes*, *resize*, *reposition*, *add*, *modify*, and *remove*. These techniques may independently impact different visualization components (e.g., *axes*, *legends*, *marks*, *labels*, and *title*), allowing for complex and varied modifications based on the device context. The modifications may apply to either

a single component, several components, or all components in the view. While most changes reflected small shifts in either layout or content, a subset of visualizations drastically changed the design (e.g., [G19, G23, G36]). The coded results are shown in Figure 8.3.

From our analysis, we found that a larger range of responsive techniques were used for the portrait orientation than the landscape orientation of a phone (Figure 8.3). For the landscape orientation, 69 of the visualizations exhibited *no changes* (29.9%) as opposed to only 6 in the portrait orientation. This difference likely reflects the fact that the landscape orientation is closer in overall shape to the screen space afforded by a desktop computer. When it comes to applying responsive techniques, we saw a variety of modifications; in particular, resizing the design often requires device-specific customizations (e.g., to *reposition* the visual content [G13, G36], *add* clarifying information [G5], *modify* annotations to change or shorten the text [G10, G13], *remove* visualization details [G13, G50, G52], or *remove* interactivity altogether [G1, G14]). Across examples in our corpus, we found that it was much more common to *remove* elements from the view (87 or 37.7%, portrait orientation) than to *add* new elements to a mobile visualization (26 or 11.3%, portrait orientation). While most designs generally employ a few minor modifications, in rare cases, authors completely redesign the visualization and/or interaction for different device contexts [G19, G23, G36].

In addition to the visual techniques, we examined the end-user interactions included in the visualizations. Most visualizations were static or did not change the core interaction type, aside from using tap rather than click. Similar to the visual techniques, many visualizations removed the interactivity completely from the mobile version rather than redesigning the interactive capabilities (e.g., [G1, G14]). However, a small subset introduced or updated the interaction to improve the experience on mobile (e.g., [G2, G23]). For example, in the New York Times article titled “Connecting the Dots Behind the 2016 Presidential Candidates” [G2], the desktop visualization includes a node-link diagram that shows the connection between individuals and organizations on mouseover. The authors updated this functionality for mobile to utilize a carousel navigation strategy (two buttons) to loop through individuals and show these same connections on a simplified version of the node-link diagram.

8.3.2 Responsive Visualization System Design Considerations

Based on our investigation of existing responsive visualizations and current design practice, we propose a new responsive visualization design system that facilitates flexible, cross-device development workflows. To realize this goal, our system adopts four key design guidelines:

1. **Enable simultaneous cross-device edits.** Simultaneous editing can accelerate iteration by reducing the time it takes to experiment with different design ideas across multiple target sizes. This capability also reduces the chance of introducing errors and inconsistencies from repeated manual application of edits.
2. **Facilitate device-specific customization.** Adaptation of the visualization content to particular device contexts is central to producing effective responsive designs. Our system therefore enables the application of device-specific customizations by focusing editing operations on a particular view or mark.
3. **Show cross-device previews.** Providing immediate, visual feedback across multiple designs allows designers to evaluate their choices in the context of all target chart sizes. Such previews help designers determine which choices should be consistent across devices and which should be customized for a particular view. Foregrounding design variation provides a complete picture of the customizations that have been applied.
4. **Support propagation of edits.** During the development process, designers may focus on refining the visualization design for one specific chart size. Techniques to propagate edits from one design to another can enable designers to quickly transfer ideas that work well for a particular size to other device contexts.

8.3.3 Responsive Visualization System

To realize these goals, we implemented a responsive visualization design tool that maintains a synchronized representation of a design across multiple target screen sizes. Figure 8.4 shows an overview of our system. The *main panel* displays a different visualization for each specified chart size. The *toolbar* and other system panels display information about the data and

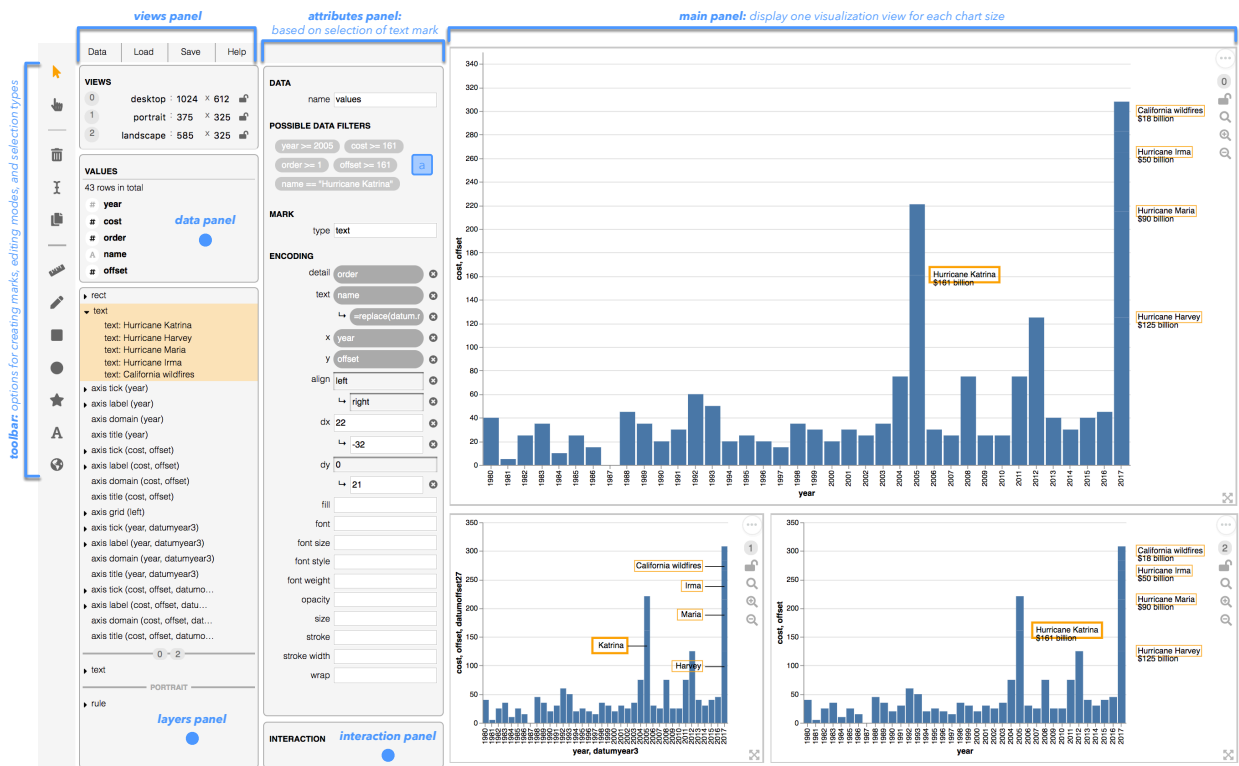



Figure 8.4: The designer creates a visualization mark by dragging a mark icon from the *toolbar* to a visualization canvas in the *main panel*. The *main panel* displays one visualization view for each device context specified by the designer. The size and name of each view is displayed in the *views panel*. The marks in the visualizations are shown in the *layers panel*. Designers can select a mark from the *layers panel* or directly on the visualization; the encodings for the mark are then displayed in the *attributes panel*. The backing data fields for the visualization are displayed in the *data panel*. To define new encodings, the designer can drag fields from the *data panel* to the *attributes panel*. Designers can specify interactive behaviors in the *interaction panel*. This figure shows the intermediate state of the responsive design process described in Section 8.4.1 with the text marks selected.

visualization components introduced for each view. Our tool supports generalized selections of visualization components both within and across views to facilitate simultaneous editing operations and customization of specific designs. Motivated by prior work [77], these generalized selections allow designers to refine the selection based on properties of the underlying data or mark encoding values. From the system panels, designers can edit and propagate customizations to multiple visualizations at once, thus reducing the need for repeated work. The *attribute panel* and *layers panel* foreground design variations between views to provide an overview of the customizations that have been applied to different designs.









In contrast to the desktop-first strategy most designers currently adopt, our system enables more flexible and iterative workflows. For example, when first developing a visualization, designers can leverage **simultaneous editing** to quickly explore the impact of high-level design decisions (e.g., overall layout or what information should be displayed) across multiple target devices. Designers can immediately **preview the design for all device contexts** while making these global edits. To resolve layout concerns for particular views, designers can **apply device-specific customizations** by selecting visualization components in a subset of views and applying local edits. Designers can also **propagate edits** to different views if a device-specific refinement works well for other device contexts. A key benefit of our system is that it allows designers to iterate fluidly back-and-forth between these global and local editing modes. The result is a more flexible workflow that promotes design exploration, view-specific adaptations, and consistency across devices. The following sections describe our system features in the context of this basic workflow.

System Startup: Viewing Multiple Device Visualizations

The *main panel* displays a blank canvas for each default chart size. Designers may import a previously constructed visualization which automatically resizes the design for these default sizes. Designers can customize the defaults to match the standard artboard sizes used by their organization. Our system automatically displays multiple, device-specific visualizations, which allows designers to **preview the design for all device contexts** and thus better incorporate considerations for the responsiveness of the design earlier in the development cycle. The view names and sizes are displayed in the *views panel*. From this panel designers can rename, resize, and create new  views at any point. When creating a new view, the system copies the design for whichever active view is closest to the new size. The system supports an arbitrary number of different views, up to the discretion of the designer. Designers may select or resize views directly from the *main panel*. The *data panel* shows the datasets and fields that have been loaded for the visualization. Each field is labeled with the automatically detected data type; clicking the data type symbol allows designers to change the type.

Rationale. One of the core challenges explored in this dissertation is to “support the tasks that matter most to the user.” Existing visualization construction systems have generally fallen short of this goal for *responsive* visualization design. As discussed in the formative interviews (Section 8.2), journalists tend to utilize a desktop-first development approach driven by system defaults rather than their personal preferences. In this system, responsive design becomes a first-class feature of the environment: multiple visualization versions are shown by default, with no implicit prioritization between designs imposed by the system.

Simultaneous Editing of the Basic Visualization Structure

To construct a new visualization, designers first create a new mark. The system uses Vega-Lite [176] as the underlying language for producing the visualizations and currently supports seven mark types: “rule” , “line” , “bar” , “circle” , “symbol” , “text” **A**, and “geoshape” . Designers create a mark by dragging the icon from the *toolbar* to any visualization canvas, or by using keyboard shortcuts to select the mark type, followed by the view. Designers can construct multiple visualizations concurrently using **simultaneous editing**. When a mark is first created it is added to all active views. By default, all views are “active.” To apply customizations, designers can focus on a single design by selecting the view number on the canvas (1) or from the *views panel*. Designers can select multiple views via locking  or unlocking  commands. Locked or “inactive” visualizations are partially grayed out.

For new marks, designers must first link the mark to one of the backing datasets in order to map data fields to the visual encodings. Similar to other *shelf construction* [175] systems, designers can drag data properties from the *data panel* to encoding shelves in the *attributes panel* to specify the visual encodings. As new encodings are specified, the system automatically adds axes and legends as appropriate. Once again, edits apply to all “active” views. When the mark has been bound to a dataset, all individual elements (e.g., all the bars in a bar chart) are placed in a group, similar to the notion of a collection in Data Illustrator [129]. The newly created group and marks are shown in the *layers panel*, which displays both the user-specified marks and marks for auto-generated guides for all views.

Rationale. To better support responsive visualization design, we raise the level of abstraction from a single visualization specification to a generalized representation that supports the design of multiple linked visualization versions. With this approach, designers can focus on the core design considerations—what data should be visualized and how—while deferring control for the adaptation of the design to the underlying system. Designers can therefore utilize their unique domain expertise to develop effective visualization designs rather than focusing on manually constructing similar designs for each device of interest.





Applying Customizations Using Flexible Selections


Designers may want to customize the visual appearance of particular marks or fully customize the visualization design for particular chart sizes. The system provides several strategies for performing flexible selections of visualization elements. Designers can select marks directly on the visualization or from the *layers panel*. When designers first click a mark, the system selects all marks in its group; designers can double click a mark to select only the particular item. Designers can toggle the selection mode from the *toolbar*. When editing visualizations, the customizations only apply to the selected marks (those highlighted in orange).

Designers can refine the selection using data filters on the mark. Data filters are displayed in the *attributes panel* (Figure 8.4a) based on the selected mark. Regardless of the selection type (group or item), the particular element that was selected acts as the anchor for the data filters; for each data field in the underlying dataset, a filter is suggested using the value of that field for the selection anchor. When selecting a data filter the system refines the selection to only the marks where the condition holds. Repeatedly selecting a filter toggles the comparison operator (e.g., \geq , \leq , $=$, and so on). In Figure 8.4 for example, the text “Hurricane Katrina” acts as the anchor for the labels; for each of the five data fields, a simple filter is created using the underlying values of this anchor point (e.g., $cost \geq 161$ and $year \geq 2005$). Selecting a different anchor changes the suggested filters (e.g., selecting “Hurricane Harvey” suggests $cost \geq 125$ and $year \geq 2017$). All of the marks in this visualization share the same backing dataset. Therefore, selecting the bar mark associated with “Hurricane Harvey” would produce the same data filters as the corresponding text mark.

Rationale. The system aims to simplify the creation of responsive visualization designs by raising the level of abstraction from a single visualization specification to the design of multiple linked versions. However, customization remains an important task for many journalists. In response, the proposed system enables users to customize the visual appearance of individual marks in the visualization. These updates can apply across visualization versions to facilitate the process of annotating and highlighting information for communicative purposes.

Enabling Customizations and Displaying Design Variation

When designers apply customizations to particular marks or views, the system helps to **show cross-device previews** by foregrounding the variation in the *layers panel* and *attributes panel*. When a mark is added or removed from a particular view, the contents of the *layers panel* are reordered to sort the elements based on which views they apply to. For example, in Figure 8.5 a separator shows that each view (“portrait” and “landscape”) has a custom axis; the marks displayed above these separators apply to all views (e.g., the two text marks). Customizations are also displayed in the *attributes panel*; in Figure 8.5, the selected “bar” mark originally had the color encoding set to “#f0f0f0” . A modification to this encoding has subsequently been applied to update the color to “firebrick” . The *attributes panel* allows designers to view design variation for the mark encodings of the currently active views. Designers can select an icon next to a customization to refine the selection to only include marks where the customization applies. For example, in Figure 8.5, clicking the arrow icon  beside the “firebrick”  customization refines the selection to only update the marks with this color when subsequent edits are applied. This functionality demonstrates the system’s support for flexible workflows; designers can edit particular customizations even when multiple views with different encodings are active.

Designers may also **propagate edits across views** by identifying and deleting design variation from the system. To propagate edits, designers can delete encodings that should no longer apply to the active visualizations. When hovering over the “delete”  symbol, the system **shows cross-device previews** of the change. This functionality helps designers

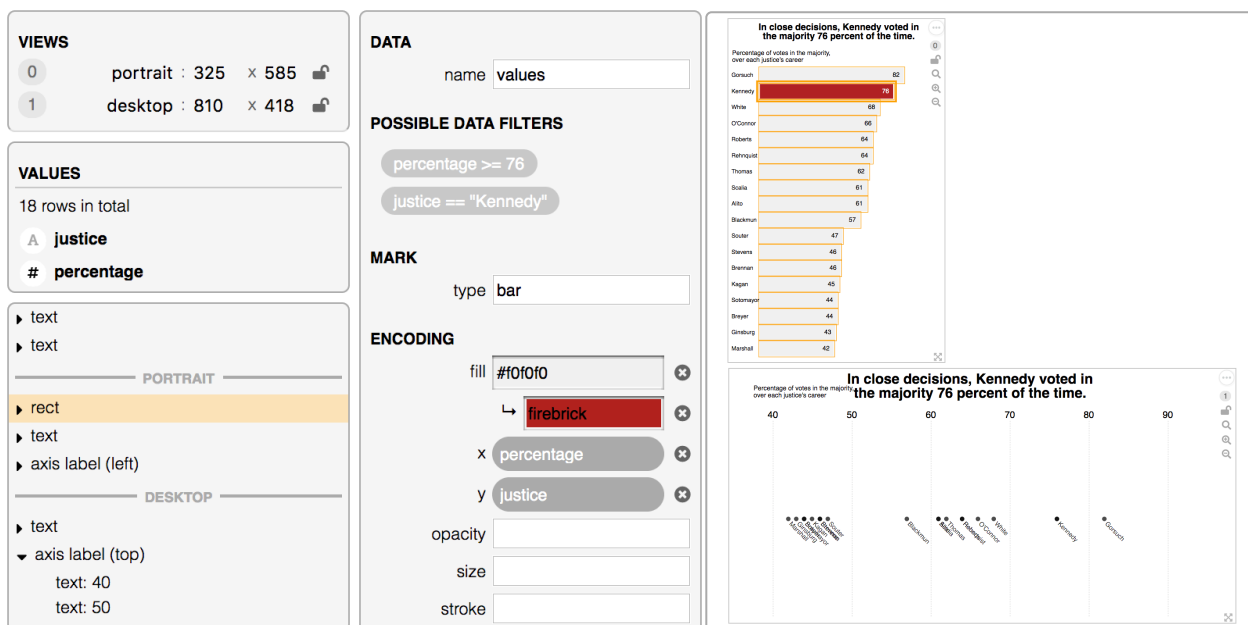



Figure 8.5: The system panels for recreating the New York Times visualization: “In close decisions, Kennedy voted in the majority 76 percent of the time” [G36]. The *layers panel* shows that while the title and subtitle appear in both views, each device has a custom set of axes and marks. The *attributes panel* shows variation in the encoding properties such as the “color,” which has been used to highlight a particular bar. Note: the panels have been resized to improve legibility of the figure.

to quickly preview the result of different modifications across multiple visualization designs and update concurrent designs with different encoding decisions. For example, in Figure 8.4, imagine that the designer wants to propagate the label simplifications in the “portrait” visualization to the “landscape” visualization. To do this, the designer marks both views as “active” (e.g., by locking  the “desktop” visualization). The user can then delete the customized encodings that are no longer desirable (e.g., `align`→`left` and `text`→`name`), which changes the alignment of the text to `right` and the text to `replace(datum.name('Hurricane', ''))` for all of the selected elements across both the “active” views.

Rationale. One of the core challenges described in this dissertation is to “communicate system behavior as actionable information.” To this end, the system emphasizes the design variation that has been applied across visualization versions and provides a direct method for removing unwanted variation. This approach enables users to form a complete picture of the customized designs and make educated decisions about how to proceed with the design process.

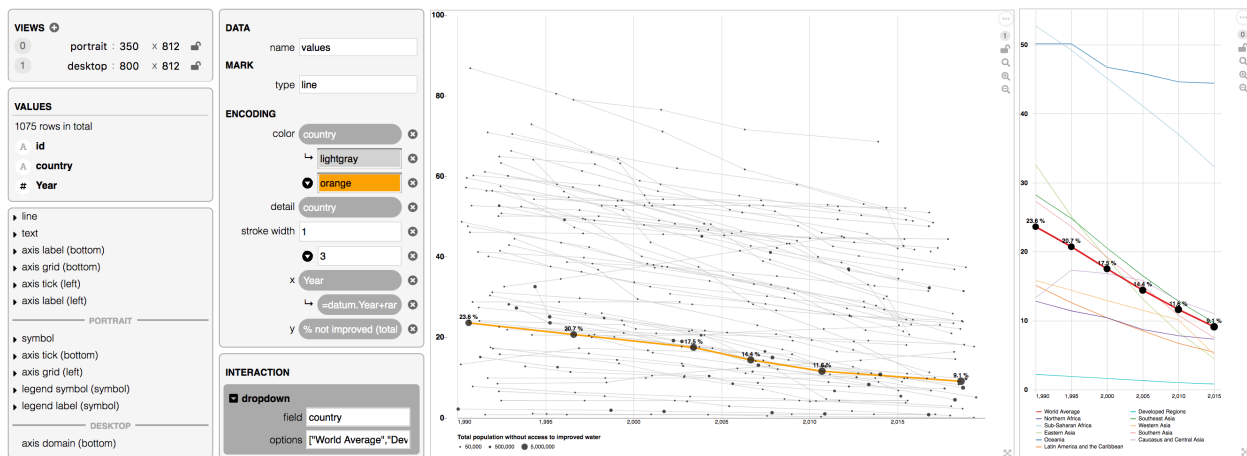



Figure 8.6: Designers may also customize the behavior of end-user interactions with the visualization. In this case, the designer specifies a dropdown interaction to update the line color and stroke width of the visualization. These customizations appear when the end user interacts with a dropdown; however, the customizations are similarly displayed alongside the other encodings and customizations in the *attributes panel*. Note: the panels have been resized to improve figure legibility.

Enabling and Customizing End-User Interactions

When a mark is selected, the *interaction panel* allows designers to specify the functionality of end-user interactions with the visualizations. Designers can first define the interaction type and then use the interaction to customize the visibility and encodings of visualization marks. Similar to other encoding customizations, when a customization applies only during interaction, it is still displayed in the *attributes panel* as a form of variation (Figure 8.6). Customizations based on end-user interaction are annotated with a symbol representing the type of end-user interaction that has been applied to the visualization element. In Figure 8.6 for example, both the color and stroke width of the line mark have been customized to respond to end-user interactions with a dropdown  to select which country should be highlighted.


Rationale. Similar to the other approaches described in this section, the functionality for specifying customizations based on end-user interaction aims to raise the level of abstraction to mirror the related functionality from other parts of the system. The customizations from interaction are similarly treated as a type of design variation and can thus utilize the same approaches to help communicate relevant and actionable information to the user.

8.4 Evaluation: Reproducing Real-World Responsive Visualizations

To demonstrate the utility of the proposed techniques for responsive visualization design, we reproduced four real-world examples using our system [G13, G36, G50, G52]. These examples illustrate four visualization types: bar chart [G13, G36], map [G52], dot plot [G36], and line chart [G50]. Each example also demonstrates a major component of our system: (1) visualization design and customization as an iterative workflow [G13]; (2) flexible data filtering for customizing annotations [G52]; (3) designing drastically different visualizations via linked editing [G36]; (4) customizing end-user interactions for different chart sizes [G50]. The following sections explore each example in terms of the design guidelines. Demo videos for each of the four walkthroughs are available on YouTube; the links are included in Appendix H. We also introduce three additional examples in Appendix H. For all seven examples, we show the different device-specific visualizations that were designed with our system.

8.4.1 An Iterative Workflow for Simplifying a Mobile Design

The New York Times article “The Places in the U.S. Where Disaster Strikes Again and Again” [G13] exhibits a variety of responsive visualization techniques to simplify the mobile version of the “Total Cost of Major Natural Disasters” visualization (Figure 8.1).

The designer starts by creating the basic visualization: a bar chart with the **year** on the x-axis and the disaster **cost** on the y-axis. Next, the designer annotates the major natural disasters contained in the data. The designer duplicates the bar mark using the “copy”  option from the *toolbar* to create a new layer with the same encodings, and customizes the mark to display the disaster name and position the marks appropriately. The designer decides to duplicate this text mark and update the text to display the disaster **cost**. Using **simultaneous cross-device edits**, the visualization is constructed for each separate view.

Using the **cross-device previews** of how this design looks for all device contexts, the designer can easily notice that the current layout is ineffective on the “portrait” orientation of the phone: the large whitespace margin wastes too much space and the bars become too

narrow. Before finalizing the design, the designer can apply **device-specific customizations** to the portrait visualization including modifying and repositioning the labels, and adding rule marks to more clearly label the bars (Figure 8.1c-e). The state of the system after customizations have been applied is shown in Figure 8.4.

After applying customizations, the designer can use **simultaneous cross-device edits** to finalize the axis styles, bar colors, and font styling. These updates apply to all views, even though customizations have been performed for the portrait visualization. Finally, the designer can add a title to the top of the view. For the desktop and landscape visualizations however, the designer might decide to reposition the title in the whitespace of the chart area instead. After making this change, the designer decides that the title was actually preferable above the chart area for the landscape visualization. To reapply this design, the designer activates both the “portrait” and “landscape” visualizations and **propagates edits** from one view to the other by deleting the customizations associated with the title.

Discussion. This example illustrates a flexible, iterative workflow in which the designer can switch between different device views to apply customizations. By maintaining links between visualization marks, the system can support global customizations even after modifications have been applied. By displaying all device views and the specific customizations to these views, the designer can maintain a complete picture of the responsive visualization design. The final visualizations produced for this example are included in Appendix H.1.

8.4.2 Applying and Refining Data-Driven Customizations

The Reuters article “Oil Spilled at Sea” [G52] includes a map visualization that reduces the number of text annotations, rescales the text, and updates the size encoding and legend for mobile visualization contexts. In our formative interviews (Section 8.2), several participants noted that maps present a particular challenge because the aspect ratio of the chart is predetermined by the map itself. One participant explained that the “*US map is a nightmare for responsiveness... you can have a very beautiful, detailed US map on the desktop but when you shrink it down to the phone you can barely see like five cities named in it*” (P2).



Figure 8.7: Recreating the map “Incidents at Sea” from Reuters Graphics [G52]. (a) The map is cluttered with many symbol and text marks; inset: text mark encodings in the *attributes panel*. (b) Removing the text marks where *size* < 252,000 emphasizes the major spills; inset: the possible and selected data filters in the *attributes panel*. (c) Reintroducing annotations for notable, historical spills provides a point of comparison; inset: the text marks as displayed in the *layers panel*.

For this visualization, the designer starts by producing the basic visualization design using **simultaneous cross-device edits**. However, this process produces a cluttered map exhibiting many overlapping labels (Figure 8.7a). To reduce this clutter, the designer can select the mark elements and use a data filter to remove elements below a particular threshold (e.g., `size < 252,000`). This action removes most of the labels in the view except for the largest ones (Figure 8.7b). However, the designer wants to compare these spills to others of historical notability. The designer can use the “annotate” **I** mode from the *toolbar* to reintroduce labels by clicking the points for the Sanchi and Exxon Valdez spills (Figure 8.7c).

Due to the shape of the map and density of the data, the designer might feel at this point that the mobile versions could be improved. The designer starts with the “landscape” phone orientation (the more natural fit for the map) and applies **device-specific customizations** to rescale and reposition the map projection to only include one of the comparison points: the Sanchi disaster. The designer also wants to include the Sanchi disaster in the “portrait” version, but due to the distance of this point from the core data, it is hard to make the visualization fit. The designer therefore decides to rotate the map for the “portrait” orientation; this change maintains the text direction of the labels for proper reading on a phone. To view the map in the unrotated orientation, the reader could change to the landscape orientation, which would adapt the view to the other version of the customized visualization.

To better support this reading experience, the designer wants to ensure that the two mobile visualizations are similar in all design decisions besides the rotation. The designer thus reactivates the “landscape” visualization, and can immediately see the **cross-device previews** in the *attributes panel*, which shows that the `scale` and `translate` properties do not match. The designer can thus **propagate edits** from the landscape visualization by deleting the undesirable customizations that were applied to the “portrait” visualization.

Discussion. Our system enables flexible selection behaviors to support customizations. The data filter selection provides a lightweight mechanism for refining selections based on the underlying data, to facilitate data-driven customizations. The “annotate” **I** mode provides

a lightweight mechanism for reincorporating deleted annotations to the view. This example also highlights the challenges with responsive visualization design for maps, but shows how a designer can mitigate this difficulty by considering the final experience of the reader. The final visualizations produced for this example are included in Appendix H.2.



8.4.3 *Producing and Reusing Radically Different Designs*

The New York Times article “With Kennedy’s Retirement, the Supreme Court Loses Its Center” [G36] includes a radical responsive redesign for the visualization: “In close decisions, Kennedy voted in the majority 76 percent of the time.” In this example, the desktop version uses a horizontal dot plot whereas the mobile version uses a vertical bar chart; each design takes advantage of the device orientation while displaying the same data.

To produce these visualizations, the designer can start by creating either the bar chart or dot plot visualization, including the marks and text labels. The designer can then focus on applying **device-specific customizations** by setting the active view and changing the underlying encodings for the mark and text (e.g., to change the mark from “bar” to “circle” and the text from **percentage** to **justice**). The designer can switch amongst the visualization views to further customize the encodings or design.

For both of these visualization versions, the designer wants to emphasize Kennedy’s position in the Supreme Court with respect to decisions by the majority and therefore customizes some of the individual marks accordingly. Down the line, the designer might decide to reuse this visualization to emphasize the position of a future justice within the larger context. To do this, the designer could swap out the underlying dataset to include updated numbers; since the customizations are applied relative to the device context and data fields, the visualization could easily adapt to a new set of data. The designer might then decide to **propagate edits** representing the highlight to a new justice in the dataset.

Discussion. While the visualizations for this example are drastically different and therefore highly customized to the device, our system can help the designer maintain a clear overall

picture of the two versions: how they vary and how they are the same. Figure 8.5 shows the state of the system panels when one of the marks has been selected. From this view, the designer can see that the mark “color” has changed from “#f0f0f0”  to “firebrick”  for some of the visualization marks. The layers panel also shows variation in which mark elements are visible on which views (e.g., to show that each device has a customized axis). The final visualizations produced for this example are included in Appendix H.3.

8.4.4 Custom End-User Interactions and Interactive Encodings

The National Geographic article “See Where Access to Clean Water Is Getting Better – and Worse” [G50] exhibits many responsive updates between the desktop and mobile version of the “Percentage of population without access to improved water” visualization. This example shows the need for custom end-user interactions: the interactive dropdown has fewer options on mobile and customized visual encodings for elements that have been interacted with.

For this example, the designer starts by customizing how the visualization should look when no interaction is applied. The designer creates the basic visualizations using **simultaneous cross-device edits** and applies **device-specific customizations** to the mobile design to remove excessive marks and update the encodings. To define the end-user interaction behavior, the designer selects the mark that should be updated during interaction, and adds a new interaction from the *interaction panel*. The designer can then specify how the end-user interaction should behave and update the mark encodings. Similar to the device-specific customizations, encodings or marks associated with an interaction are displayed in the hierarchical structure of the system. Figure 8.6 shows the state of the system panels after defining a dropdown interaction to select a line and display a corresponding text mark.

Discussion. Design variation from end-user interaction is similarly displayed alongside other encoding modifications within the system to provide **cross-device previews**. The *attributes panel* and *layers panel* provide a clear overview of all encoding decisions for a particular mark, allowing the designer to easily update encodings for any use case throughout the system. The final visualizations produced for this example are included in Appendix H.4.

8.5 Limitations and Future Work

We built our system on top of Vega-Lite [176] to leverage its ability to express parameterized graphical elements (marks and axes) that exhibit reasonable default behavior when scaled to different chart sizes. While this decision reduced the effort required to implement a functional visualization design tool, the capabilities of our current prototype are coupled with the underlying representation, sometimes resulting in awkward user experiences. For example, axis labels and text marks exhibit different sets of editable properties due to differences in the Vega-Lite specification. Furthermore, elements in one view only match those in other views if they were specified simultaneously. Future work should better decouple the front end from low-level details of the core visualization machinery. Matches could be computed based on the visual similarity and underlying data, rather than the internal structure.

The current system allows designers to specify an arbitrary number of views based on their design needs. However, for real-world use cases designers generally focus on a limited set of artboard sizes as defined by their organization. Therefore, our examples similarly use three or four views targeting different device sizes. While it is possible to create more views, there are limitations on the amount of screen space available for development and the number of views that a designer could feasibly view or comprehend at one time. Future work should explore the development patterns of designers and how they would work with multiple views simultaneously. New techniques to cluster and summarize views could prove useful for alleviating challenges that arise when working with a larger space of designs.

Due to a hesitance from our interview participants regarding automatic techniques, we chose to focus on an ad hoc, user-driven visualization design, which does produce a largely manual process. Exploring new techniques to increase the number of views or better summarize the space of designs could help designers transition to more automated or dynamic procedures. Such automation could be particularly useful for more general responsive use cases such as the design of dashboards [171]. Future work should explore how best to ensure the transparency of automation within the development process, as related to the challenge that systems should “communicate system behavior as actionable information” to the user.

One approach to the automation of visualization designs could be to incorporate constraints that encode the desired customizations for different visualization versions. Related to the discussion from Chapter 3, this approach introduces a number of new considerations about how best to support users in authoring and debugging the behavior of constraints, and further raises a question about the extent to which this functionality should be communicated to the user. Charticulator [161] has begun to explore this space by utilizing constraints as the underlying layout engine for a visual builder system for bespoke visualization design (see Section 2.1.2); however, these constraints are hidden from the end user and can therefore prove challenging to understand or debug. New systems for visualization design utilizing constraints should therefore explore trade-offs about the level of abstraction employed in the system and new strategies for debugging and program understanding.

Our system includes a variety of interaction techniques such as keyboard shortcuts, toolbar menus, and different types of mouse clicks for refining selections. Since generalized selections are a key part of our proposed workflow, it would be valuable to refine the usability and performance of such interactions. In particular, updating the data filter procedure to provide more intricate or generalizable suggestions could improve the overall utility. There are also opportunities to explore new ways for designers to propagate customizations across views, perhaps through analysis of how the graphics themselves are arranged in each visualization.

In this work, we reproduce four real-world examples [G13, G36, G50, G52] to demonstrate the benefits of our system for responsive visualization design. To be clear, the procedures presented do not reflect the actual design process for the published visualizations; instead, they highlight a set of flexible, iterative workflows that contrast the more linear design process that designers typically adopt. Future work should explore the nuances of this system and the overall impact on the design of responsive visualizations in practice.

8.6 Summary of Contributions

This chapter explores the design of responsive visualizations that adapt the visualization design based on the screen size or interactive capabilities of the device. This chapter contributes a survey of 231 responsive visualizations from twelve news organizations to examine existing responsive design practices. This chapter further contributes formative interviews with five authors about their design process and rationale. From these interviews we found that designers generally leverage a linear, desktop-first development process while largely considering the mobile implications in the abstract. Based on the formative interviews and survey results, we identified four design guidelines and contribute a set of core system features to support responsive visualization design. Our system displays multiple views for different device contexts and foregrounds design variation to provide a complete picture of the responsive techniques applied. Designers can construct visualizations using both simultaneous global edits or local customizations to the designs. To demonstrate the utility of these techniques for more flexible development workflows, we reproduced four real-world examples selected from our earlier survey and described the relevant system functionality.

In contrast to the existing linear workflows described in the formative interviews, these examples demonstrated the expressiveness and flexibility of our system for supporting the iterative design of responsive visualizations. This process allows designers to focus on important considerations around how the design should look for each device context, while reducing the burden of creating multiple designs and propagating changes amongst designs manually. Our techniques to surface the design variation can also help designers better understand the set of modifications that have been made and highlight potential disconnects that might otherwise be overlooked. Finally, this approach demonstrates opportunities for visualization reuse by allowing designers to swap out the data while maintaining the customized views.

This work was done in collaboration with Wilmot Li and Zhicheng Liu from Adobe Research, and was originally published at CHI 2020 [84].

Chapter 9

CONCLUSION

In this dissertation, I contribute the design of new programming languages and program visualization tools for constructing interactive visualizations. In particular, this work explores the design of custom graph layouts (Chapter 4), interactive visualizations (Chapter 5-7), and responsive visualizations (Chapter 8). Across these projects, my research aims to better understand people and to help people better understand systems. For each project, I first sought to understand the pain points faced by individuals in their current development process (Section 3.2, Section 5.4, Section 8.2). These interviews surfaced concerns specific to each domain, as well as aspects of the three high-level challenges explored in dissertation.

These challenges complicate the design of new programming languages and program understanding techniques, while also suggesting exciting opportunities for future research. The first challenge is to “raise the level of abstraction to reflect user expertise.” This challenge is a central guiding principle behind the work explored in this dissertation, but the main challenge here lies in identifying the appropriate level of abstraction to employ. When abstracting the underlying system behavior, low-level system details become less useful for end-user programmers. The second challenge is therefore to “communicate system behavior as actionable information” to the user. An important part of this process is not only to identify how best to communicate information, but also what information *not* to show to the user. To this end, the third challenge is to “support the tasks that matter most to the user.” When raising the level of abstraction and communicating system details, it is important for systems to do so in such a way that users can focus on the most important development tasks while reducing the burden for less essential operations. This requirement can be further complicated when acknowledging that users often have changing, and sometimes conflicting, needs.

Motivated by these challenges, I identified and developed new ways in which to improve the development process by designing systems that allow users to focus on the high-level constructs with which they are most familiar, as well as their preferences regarding their primary tasks (Section 4.2, Section 6.1.1, Section 7.1, Section 8.3.3). To better understand the impact and utility of these systems, I then evaluated the proposed techniques via user studies (Section 6.2, Section 7.3) or by reproducing real-world examples (Section 4.3, Section 8.4). While these projects contribute novel advances for visualization design and program understanding, they also surface exciting new areas for future work (Section 3.3, Section 4.4, Section 6.3, Section 7.4, Section 8.5, Section 9.3). To conclude this dissertation, I reflect on the three core challenges and their relationship to the work presented in this dissertation (Section 9.2). I then wrap up this dissertation with some short concluding remarks.

9.1 Summary of Contributions

In Chapter 3, I contribute a set of preliminary interviews with programmers about the utility and program understanding challenges of constraints in end-user systems. I then explore the use of constraints for customized graph layout in Chapter 4, and contribute SetCoLa: a new domain-specific language for customized graph layout using constraints. While SetCoLa can reduce the number of user-authored constraints by one to two orders of magnitude, this approach surfaces program understanding challenges for declarative programming languages.

In Chapter 5, I introduce Vega [177] as a platform on which to explore new opportunities to improve program understanding for declarative languages. To this end, I contribute a prototype data flow graph visualization of the underlying Vega runtime, and interviews with expert Vega users about the utility of this approach. Participants felt that while the data flow graph can prove useful for system developers, it provides too much information tangential to their end-user debugging needs. To better support program understanding for end-user programmers, I contribute a set of visual debugging techniques for reactive data visualization in Chapter 6. These techniques raise the level of abstraction for program understanding tools to communicate relevant system details in terms of constructs with which users are familiar.

In an evaluation, these techniques helped participants trace errors through unfamiliar code, but sometimes required users to already know where to look across separate system panels. To reduce the gap between the program understanding techniques and user-authored code, Chapter 7 contributes a design space of embedded visualizations that appear directly inline in the code. This approach facilitates code authoring while surfacing relevant system details.

Finally, whereas Vega supports the design of a single interactive visualization, designers sometimes need to develop responsive visualizations that adapt the content based on the screen size or interactive capabilities of the viewer’s device. To this end, Chapter 8 contributes a survey of existing responsive visualization techniques used in news articles, and interviews with journalists about their design and development process. This chapter further contributes a set of design guidelines and core system features inspired by the survey and interviews that promotes responsive visualization design via more flexible development workflows. To demonstrate the utility of these features, I reproduced four real-world examples to highlight the expressiveness and flexibility for responsive visualization design.

9.2 Discussion and Reflections on Three Core Dissertation Challenges

Chapter 1 introduces three core challenges that are central to the work described throughout this dissertation and are motivated by the thesis statement: “the design of new languages and program visualization tools that raise the level of abstraction from low-level system details to domain-specific concepts and operations for interactive visualization design can help end-user programmers more effectively *author*, *understand*, and *reuse* both code and data.” The challenges are as follows: (1) raise the level of abstraction to reflect user expertise; (2) communicate system behavior as actionable information; and (3) support the tasks that matter most to the user. While the challenges themselves intuitively reflect concepts from human-computer interaction research, the techniques required to realize each goal are not immediately apparent or straightforward. In the following sections, I reflect on these challenges and the approaches employed in my dissertation work. This discussion helps to illustrate my overall research process and motivate new areas for future research (Section 9.3).

Challenge 1: Raise the level of abstraction to reflect user expertise.

The goals and intent of the user are the primary driving factors for any system behavior, so it is essential that systems prioritize the user’s unique position. By ensuring that systems reflect the user’s expertise, systems can better support program understanding and allow users to effectively employ their expertise beyond concerns for the system-level implementation.

This challenge often arose during our interviews with both expert and novice users. In our formative interviews on understanding constraints (Chapter 3), participants noted that while constraints can provide a useful way to encode domain knowledge, for end-user facing systems *“designers don’t necessarily want to think in that way. So I’m not even telling them they’re constraints anymore.”* Participants from our formative interviews on visual debugging techniques (Chapter 5) expressed similar concerns, this time from the end-user side: *“the [data flow] graph presumes insight into how Vega’s internals operate.”* Participants explained that while the data flow graph visualization can be helpful for system developers, assuming that end users will be familiar with the low-level implementation constructs does not appropriately reflect their needs for end-user program understanding tasks. To address these concerns, this dissertation contributes SetCoLa (Chapter 4) and new automatic visualization techniques (Chapter 5-7) for Vega [177]. These approaches help end-user programmers better navigate and understand code by allowing users to focus on the domain-specific concepts and operations of interest rather than the low-level system details. To accomplish this goal, these techniques aim to reduce the level of separation between the code the end-user programmer writes, and the tools used to understand the underlying system behavior.

As described in Chapter 4, SetCoLa is a high-level language for designing customized, domain-specific graph layouts by leveraging the domain expertise of the user. SetCoLa allows the user to design custom layouts using a simple set of constraints relative to the particular, domain-specific properties of interest. This approach reduces the number of user-authored constraints by one to two orders of magnitude and further supports reuse of the layout across multiple graphs in the same domain. For this project, the user can focus primarily on how the layout should encode domain properties based on the user’s unique domain knowledge.

The proposed visual debugging techniques (Chapter 6-7) emphasize Vega’s high-level concepts and are thus at the appropriate level of abstraction for end-user programmers. The visualizations focus on the signal values which encapsulate the interaction logic for interactive visualizations and the data properties that drive these visual encoding decisions. In evaluations of these techniques, we showed that novice programmers were able to accurately trace errors through unfamiliar code (Chapter 6) and better answer program understanding questions (Chapter 7) when supported by visualizations of the program behavior. As one participant explained: *“the code visualizations helped better locate the signals and made me more confident about my answers.”* These visualizations allow users to more effectively navigate the code and have a better overall understanding of the program behavior.

While Vega focuses on the design of a single visualization, journalists must often produce multiple designs to support responsive adaptation of the content for different devices. When reflecting on the process of creating multiple responsive designs, one journalist explained that when producing the separate designs, *“it feels like a chore... You want to be working on the story; you want to not be working on polishing things for small audiences.”* To this end, our proposed responsive visualization system (Chapter 8) raises the level of abstraction from a single visualization specification to support the concurrent design of multiple customized visualization versions. Utilizing this approach, designers can develop visualizations using simultaneous editing; components across views are linked so as to reduce the specification effort and allow users to focus on authoring and reusing the design, rather than requiring users to manage manual changes to individual visualization versions.

Challenge 2: Communicate system behavior as actionable information.

In order to improve how people interact with systems, systems should communicate information about the behavior in ways that are immediately actionable by the user. For programming contexts, an important part of understanding an error is to identify what the error is, when it occurs, and how that behavior relates to the original source code. For my projects exploring visual debugging techniques for Vega (Chapter 5-7), the system provides a direct link

between the debugging properties and the corresponding components in the source code. As one participant from our evaluation noted: *“the [in situ] visualizations allowed me to connect the dots between the code, its properties, and what it did.”* Another participant explained that *“I found it helpful to be able to interact with the data on a graphical, physical level.”* In many programming or data science pipelines, the data plays an essential but sometimes hidden role in the analysis process. While techniques to inspect the data are common, it is not always clear exactly how to use the information that is gleaned when simply printing out rows in a table. In our approach, by reducing the separation between the code and the availability of the program understanding tools, users can better focus on authoring new code and understanding the data responsible for driving the overall program behavior.

For our responsive visualization system (Chapter 8), we similarly provided contextually relevant information that could support users by steering their interactions with the system. To do this, we display design variation as a first-class component of our system panels; for example, users can immediately see which visualization marks have been modified or exist on particular versions but not others. Users therefore have an immediate way in which to identify incorrect or unnecessary variation and better standardize the views. This system also ensures that the target of any edits has been clearly identified prior to a change; marks or views that will be updated are highlighted whereas those outside the selection are deemphasized.

Finally, SetCoLa presents a particular challenge for how best to communicate the system behavior (Chapter 4). In SetCoLa, users focus on writing a small series of constraints that encode their domain knowledge, which compiles into one to two orders of magnitude more low-level constraints for the underlying layout engine. To support program understanding, SetCoLa labels all system generated constraints with the source information and displays details of the conversion in the system view. With this approach, users can inspect how sets are created by highlighting the constituent nodes and view the corresponding low-level constraints. An advantage of SetCoLa is that constraints reflect domain-specific details, so the user can employ this information to better understand when particular sets or generated constraints break the original expectations for how the domain layout should be computed.

Challenge 3: Support the tasks that matter most to the user.

This challenge is particularly difficult because it can be hard to correctly identify what task is most essential to the user, especially given that users may have ever-changing needs. Consider our work on the design of responsive visualizations (Chapter 8). Journalists are constantly developing responsive visualizations to display content for news articles: *“it is something that needs to be done for every single graphic.”* However, journalists are currently undersupported in completing this highly essential task by the range of tools that are currently available. When reflecting on the predominantly linear and desktop-first development process, one journalist noted that *“by virtue of sort of sketching graphics on my laptop or on my desktop screen, often the first iteration of something works best at those screen widths.”* An important takeaway from these interviews is that system defaults can have a major impact on how the user will engage with the system. This impact can even be strong enough to override the primary goals of the user (responsive mobile designs) to enforce the system defaults. Another journalist shared a similar observation about the desktop-first development process, but more explicitly called out the role of the system in this pipeline: *“much of the programs we use are geared towards desktop first or feel that way, anyway, so if all of them had a slight shift in default or in tone I feel like that would also help us to think that way.”* To this end, when developing our prototype system for responsive visualization design (Chapter 8) it was therefore essential for the system to reflect the flexible workflows that journalists wanted to follow. As suggested by our interview participants, we developed a visualization construction system that represents all device contexts that designers are interested in. With this workflow, journalists can see and explore how visualizations will look across devices rather than dismissing ideas based on their presumptions of how it might work on mobile.

Our earlier work on visual debugging techniques observed a similar trajectory in the evolution of our approach (Chapter 5-7). Our first prototype data flow graph visualization provided an accurate representation of the system behavior, with too much information tangential to end-user debugging needs (Chapter 5). While our follow-up set of visual debugging techniques were able to effectively help users trace errors back to the original code, these

techniques employed a series of separate but coordinated views to display the useful program visualizations (Chapter 6). In the evaluation, we found that participants often overlooked informative views that were on the periphery of their attention or were otherwise hidden from view. When reflecting on the low score provided for one of the proposed techniques, one participant noted that the system should *“promote its appearance more.”* Similar to this approach, many existing debugging tools present program understanding techniques that must be explicitly invoked and appear in separate, coordinated views. However, this process imposes a burden on users required to now seek out the information of interest. To better proactively support users during their primary development task (e.g., code authoring or testing) we devised a series of in situ code augmentations that appear directly inline in the source code (Chapter 7). These visualizations provide insightful slices of the program behavior without requiring users to stop their main development task to explicitly seek this information out. In support of this approach, one participant explained that *“the biggest factor for me was just seeing which values change in real-time when interacting with the visualization.”* These visualizations can better attract attention to relevant changes in the code and provide contextually appropriate information based on the user’s current focus.

The SetCoLa project was directly motivated by the need to better support how domain experts author customized graph layouts (Chapter 4). Existing techniques tend to require extensive programming expertise and user effort to develop customized layout algorithms or tools. We therefore sought to better support this development task by focusing the implementation on the unique domain knowledge of the user. In addition to creating a custom layout for a particular graph, it was important for these layouts to be extensible to other graphs in the same domain to limit the amount of specification effort required by the domain expert to reuse the layout. SetCoLa effectively realizes these goals by allowing users to specify custom layouts with one to two orders of magnitude fewer constraints than required by the underlying constraint engine. We further demonstrated how these layouts can be reapplied across any number of graphs exhibiting the same set of domain-specific properties.

9.3 Future Research Directions

Motivated by three core challenges, this dissertation contributes new programming languages and program visualization tools for constructing interactive visualizations. This work particularly focuses on understanding and improving how users interact with systems for visualization design. However, there are many interesting avenues for future research in this space and beyond. In particular, future work should continue to explore how to ease the burden on people and empower them to focus on the applications and designs that matter most. Towards this goal, future work should explore how novel methods and tools might offset the burden on users while adapting to their changing needs and available resources.



My work on visualizations for code understanding [87, 88] exemplifies how such techniques may be applied for a large class of *reactive* programming languages, but future work should explore how to effectively incorporate real-time program visualizations into *imperative* programming domains or other data-centric end-user systems. For example, while SetCoLa [83] aims to simplify the process of creating customized graph layouts, the results from constraint satisfaction can still be hard to comprehend; many challenges remain around understanding why a satisfying set of constraints is not achieving the desired layout, or deciding which constraints need to be added, changed, or removed to accomplish the intended result. Future research should therefore explore new techniques to facilitate program understanding for constraints, which remains a largely open area for novel research contributions. Constraints may also prove useful for reflecting the expectations of the user for responsive visualization design [84], and similarly raises issues of program understanding and debugging as explored in my prior work. An open challenge in this space is to further explore options for automatically adapting the visualization content based on device or user context. As the number of visualizations grows, new techniques are required to group and summarize multiple designs, and support designers in exploring the space of visualizations that they create.

The second challenge discussed in this dissertation is to “communicate system behavior as actionable information.” This challenge raises many questions about what information is most important and how best to communicate such information to end users. To facilitate the

process of interacting with complex end-user systems, future work should explore how best to communicate the user’s intent in the system and translate the system output into actionable information. This future work could prove integral to the application and widespread use of constraints. Furthermore, these challenges and research questions can apply to a wider range of systems. Future work should therefore continue to explore how communicating both user and system intent can positively impact a larger class of end-user systems.

One common assumption across my research projects to date is that the data will arrive in a clean, ready-to-use format. However, such assumptions rarely hold in real-world workflows. Future work should therefore explore how best to incorporate data cleaning or data transformation alongside existing design and development processes. Related work has begun to explore new techniques to support the initial data wrangling step [40, 68, 72, 101, 191]. For example, Wrangler [101] enables users to interactively specify and preview data transformations directly on a data table to produce reusable data wrangling scripts. More recently, Wrex [40] aims to support data scientists in authoring data transformation code via programming-by-example for computational notebooks. While powerful, these techniques also require a fully-formed, albeit imperfect, data table. Recent work has also explored strategies to support the extraction of tabular data [29, 30, 31, 74, 85]. Rousillon [29] is a tool for easily scraping hierarchical data from the web using programming-by-demonstration. My prior work has also explored automatic extraction and analysis of tabular data from PDFs, as well as interactive techniques for users to repair structural errors in such tables [85]. However, an important part of effective data wrangling is the ability to easily integrate the process into developers’ existing workflows. Kandel et al. [100] describe several exciting research directions for incorporating visualization into the data wrangling process, and further note the importance of developing new integrated wrangling and analysis tools. Future work should therefore explore new strategies for debugging and employing data transformations during other steps of the analysis process, and new ways to better support iterative refinement of the underlying data.

Many data transformations are possible in Vega [177], but there is limited support for authoring or debugging these transformations. My work on program understanding systems

aims to provide insight into how these transformations behave or what happens when they fail [87, 88]. For example, visual encodings, particularly those involving scales, are often difficult to debug or conceptualize; these mappings prove especially challenging to understand in end-user systems for which the programmatic component is sometimes deemphasized. Satyanarayan et al. [175] discuss the challenges and design trade-offs regarding the visibility of scales for three different visual builders: Lyra [174], Data Illustrator [129], Charticator [161]. However, future work should explore new techniques to facilitate the program understanding process of such transformations. My work on visual debugging techniques proposes one approach (Chapter 6): users can probe points in the output visualization to see how data maps to the visual encodings of the marks (e.g., the color) as a tooltip  directly on the output visualization [87]. While interpreting the behavior of scales is one source of difficulty, the underlying data and behavior of data transformations is also difficult to understand. In my follow-up work (Chapter 7), users may view code-embedded visualizations to see the variation in a dataset  based on the behavior of data transformations over time [88]. Both of these approaches surface details of the otherwise opaque data processing pipeline in Vega. While Vega acts as an exemplar of a larger class of reactive programming languages, future work should explore how these techniques or new approaches can better support users in developing and integrating data transformations into their development processes for different end-user systems and data analysis workflows.

9.4 Concluding Remarks

Visualizations play an important role in the analysis and communication of data. Yet existing strategies for how people design and develop customized, interactive visualizations presents unique challenges for how best to employ the expertise of the user and achieve their development goals. In this dissertation I contribute new techniques to support users in authoring and reusing customized visualizations by helping them better understand the behavior of the system at hand. I believe that these approaches can help inform future work to develop new systems that adapt based on the user’s intent, and ever-changing goals and resources.

BIBLIOGRAPHY

- [1] Keith Andrews. 2018. Responsive Visualisation. In *CHI Workshop on Data Visualization on Mobile Devices (MobileVis)*.
https://mobilevis.github.io/assets/mobilevis2018_paper_4.pdf
- [2] Apple Inc. 2018. Understanding Auto Layout.
<https://developer.apple.com/library/archive/documentation/UserExperience/Conceptual/AutolayoutPG/index.html>. (2018). Accessed: 2020-05-26.
- [3] David L Atkins, Thomas Ball, Glenn Bruns, and Kenneth Cox. 1999. Mawl: A Domain-Specific Language for Form-Based Services. *IEEE Transactions on Software Engineering* (1999). <https://doi.org/10.1109/32.798323>
- [4] Greg J Badros, Alan Borning, and Peter J Stuckey. 2001. The Cassowary Linear Arithmetic Constraint Solving Algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)* (2001). <https://doi.org/10.1145/504704.504705>
- [5] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A Survey on Reactive Programming. *ACM Computing Surveys (CSUR)* (2013).
<https://doi.org/10.1145/2501654.2501666>
- [6] Aaron Barsky, Tamara Munzner, Jennifer Gardy, and Robert Kincaid. 2008. Cerebral: Visualizing Multiple Experimental Conditions on a Graph with Biological Context. *IEEE Transactions on Visualization & Computer Graphics* (2008).
<https://doi.org/10.1109/TVCG.2008.117>
- [7] Edward B Baskerville, Andy P Dobson, Trevor Bedford, Stefano Allesina, T Michael Anderson, and Mercedes Pascual. 2011a. Spatial Guilds in the Serengeti Food Web Revealed by a Bayesian Group Model. *PLoS Computational Biology* (2011).
<https://doi.org/10.1371/journal.pcbi.1002321>
- [8] Edward B Baskerville, Andy P Dobson, Trevor Bedford, Stefano Allesina, T Michael Anderson, and Mercedes Pascual. 2011b. Interactive Serengeti Food Web.
<http://edbaskerville.com/research/serengeti-food-web/groups-figure3-interactive/>. (2011). Accessed: 2020-04-21.

- [9] Kayce Basques. 2019a. JavaScript Debugging Reference. <https://developers.google.com/web/tools/chrome-devtools/javascript/reference>. (2019). Accessed: 2020-04-29.
- [10] Kayce Basques. 2019b. Simulate Mobile Devices with Device Mode in Chrome DevTools. <https://developers.google.com/web/tools/chrome-devtools/device-mode/>. (2019). Accessed: 2020-04-26.
- [11] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. 2009. Gephi: An Open Source Software for Exploring and Manipulating Networks. *International AAAI Conference on Weblogs and Social Media* (2009). <http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154>
- [12] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. 1998. JTS: Tools for Implementing Domain-Specific Languages. In *Proceedings of the 5th International Conference on Software Reuse*. <https://doi.org/10.1109/ICSR.1998.685739>
- [13] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G Tollis. 1998. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall PTR.
- [14] Fabian Beck, Fabrice Hollerich, Stephan Diehl, and Daniel Weiskopf. 2013a. Visual Monitoring of Numeric Variables Embedded in Source Code. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. <https://doi.org/10.1109/VISSOFT.2013.6650545>
- [15] Fabian Beck, Oliver Moseler, Stephan Diehl, and Günter Daniel Rey. 2013b. In Situ Understanding of Performance Bottlenecks Through Visually Augmented Code. In *2013 21st International Conference on Program Comprehension (ICPC)*. <https://doi.org/10.1109/ICPC.2013.6613834>
- [16] Moritz Y Becker and Isabel Rojas. 2001. A Graph Layout Algorithm for Drawing Metabolic Pathways. *Bioinformatics* (2001). <https://doi.org/10.1093/bioinformatics/17.5.461>
- [17] Richard A Becker and William S Cleveland. 1987. Brushing Scatterplots. *Technometrics* (1987). <https://www.jstor.org/stable/1269768>
- [18] Austin R Benson, David F Gleich, and Jure Leskovec. 2016. Higher-Order Organization of Complex Networks. *Science* (2016). <https://doi.org/10.1126/science.aad9029>

- [19] Alex Bigelow, Carolina Nobre, Miriah Meyer, and Alexander Lex. 2019. Origraph: Interactive Network Wrangling. In *IEEE Conference on Visual Analytics Science and Technology (VAST)*. <https://doi.org/10.1109/VAST47406.2019.8986909>
- [20] Alan Borning. 1981. The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory. In *ACM Transactions on Programming Languages and Systems*. <https://doi.org/10.1145/357146.357147>
- [21] Mike Bostock. 2014. Visualizing Algorithms. <https://bost.ocks.org/mike/algorithms/>. (2014). Accessed: 2020-04-21.
- [22] Michael Bostock and Jeffrey Heer. 2009. Protovis: A Graphical Toolkit for Visualization. *IEEE Transactions on Visualization & Computer Graphics* (2009). <https://doi.org/10.1109/TVCG.2009.174>
- [23] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D3: Data-Driven Documents. *IEEE Transactions on Visualization & Computer Graphics* (2011). <https://doi.org/10.1109/TVCG.2011.185>
- [24] Nadieh Bremer. 2019. Techniques for Data Visualization on both Mobile & Desktop. <https://www.visualcinnamon.com/2019/04/mobile-vs-desktop-dataviz>. (2019). Accessed: 2020-04-21.
- [25] Karin Breuer, Amir K Foroushani, Matthew R Laird, Carol Chen, Anastasia Sribnaia, Raymond Lo, Geoffrey L Winsor, Robert EW Hancock, Fiona SL Brinkman, and David J Lynn. 2012. InnateDB: Systems Biology of Innate Immunity and Beyond—Recent Updates and Continuing Curation. *Nucleic Acids Research* (2012). <https://doi.org/10.1093/nar/gks1147>
- [26] Brian Burg, Richard Bailey, Amy J Ko, and Michael D Ernst. 2013. Interactive Record/Replay for Web Application Debugging. In *ACM User Interface Software & Technology (UIST)*. <https://doi.org/10.1145/2501988.2502050>
- [27] Luca Cardelli and Rowan Davies. 1999. Service Sombinators for Web Computing. *IEEE Transactions on Software Engineering* (1999). <https://doi.org/10.1109/32.798321>
- [28] Bay-Wei Chang, Jock D Mackinlay, Polle T Zellweger, and Takeo Igarashi. 1998. A Negotiation Architecture for Fluid Documents. In *ACM User Interface Software & Technology (UIST)*. <https://doi.org/10.1145/288392.288585>

- [29] Sarah E Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *ACM User Interface Software & Technology (UIST)*. <https://doi.org/10.1145/3242587.3242661>
- [30] Zhe Chen and Michael Cafarella. 2013. Automatic Web Spreadsheet Data Extraction. In *Proceedings of the 3rd International Workshop on Semantic Search over the Web*. <https://doi.org/10.1145/2509908.2509909>
- [31] Zhe Chen and Michael Cafarella. 2014. Integrating Spreadsheet Data via Accurate and Low-Effort Extraction. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. <https://doi.org/10.1145/2623330.2623617>
- [32] Joel E Cohen, Tomas Jonsson, and Stephen R Carpenter. 2003. Ecological Community Description using the Food Web, Species Abundance, and Body Size. *Proceedings of the National Academy of Sciences* (2003). <https://doi.org/10.1073/pnas.232715699>
- [33] Matt Conlen and Jeffrey Heer. 2018. Idyll: A Markup Language for Authoring and Publishing Interactive Articles on the Web. In *ACM User Interface Software & Technology (UIST)*. <https://doi.org/10.1145/3242587.3242600>
- [34] Matt Conlen, Alex Kale, and Jeffrey Heer. 2019. Capture & Analysis of Active Reading Behaviors for Interactive Articles on the Web. *Computer Graphics Forum (Proc. EuroVis)* (2019). <https://doi.org/10.1111/cgf.13720>
- [35] Joseph Cottam and Andrew Lumsdaine. 2008. Stencil: A Conceptual Model for Representation and Interaction. In *Information Visualisation*. <https://doi.org/10.1109/IV.2008.66>
- [36] Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. *ACM SIGPLAN Notices* (2013). <https://doi.org/10.1145/2499370.2462161>
- [37] Camil Demetrescu, Irene Finocchi, and John T Stasko. 2002. Specifying Algorithm Visualizations: Interesting Events or State Mapping? In *Software Visualization*. https://doi.org/10.1007/3-540-45875-1_2
- [38] Google Developers. 2019. ConstraintLayout. <https://developer.android.com/reference/android/support/constraint/ConstraintLayout>. (December 2019). Accessed: 2020-04-22.

- [39] Steven P Dow, Alana Glassco, Jonathan Kass, Melissa Schwarz, Daniel L Schwartz, and Scott R Klemmer. 2010. Parallel prototyping leads to better design results, more divergence, and increased self-efficacy. *ACM Transactions on Computer-Human Interaction (TOCHI)* (2010). <https://doi.org/10.1145/1879831.1879836>
- [40] Ian Drosos, Titus Barik, Philip J Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. In *ACM Human Factors in Computing Systems (CHI)*. <https://doi.org/10.1145/3313831.3376442>
- [41] Tim Dwyer. 2020. cola.js: Constraint-Based Layout in the Browser. <http://marvl.infotech.monash.edu/webcola/>. (2020). Accessed: 2020-04-22.
- [42] Tim Dwyer and Yehuda Koren. 2005. Dig-CoLa: Directed Graph Layout through Constrained Energy Minimization. In *IEEE Symposium on Information Visualization (InfoVis 2005)*. <https://doi.org/10.1109/INFVIS.2005.1532130>
- [43] Tim Dwyer, Yehuda Koren, and Kim Marriott. 2006. IPSep-CoLa: An Incremental Procedure for Separation Constraint Layout of Graphs. *IEEE Transactions on Visualization & Computer Graphics* (2006). <https://doi.org/10.1109/TVCG.2006.156>
- [44] Tim Dwyer and Kim Marriott. 2007. Constrained Stress Majorization using Diagonally Scaled Gradient Projection. In *International Symposium on Graph Drawing*. https://doi.org/10.1007/978-3-540-77537-9_23
- [45] Tim Dwyer, Kim Marriott, and Michael Wybrow. 2008a. Dunnart: A Constraint-Based Network Diagram Authoring Tool. In *International Symposium on Graph Drawing*. https://doi.org/10.1007/978-3-642-00219-9_41
- [46] Tim Dwyer, Kim Marriott, and Michael Wybrow. 2008b. Topology Preserving Constrained Graph Layout. In *International Symposium on Graph Drawing*. https://doi.org/10.1007/978-3-642-00219-9_22
- [47] Tim Dwyer and George Robertson. 2009. Layout with Circular and Other Non-Linear Constraints using Procrustes Projection. In *International Symposium on Graph Drawing*. https://doi.org/10.1007/978-3-642-11805-0_37
- [48] Tim Dwyer and Michael Wybrow. 2018. libcola — Overview. <http://www.adaptagrams.org/documentation/libcola.html>. (2018). Accessed: 2020-04-22.

- [49] Peter Eades, Carsten Gutwenger, Seok-Hee Hong, and Petra Mutzel. 2009. Graph Drawing Algorithms. In *Algorithms and Theory of Computation Handbook*.
- [50] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. 2001. Graphviz—Open Source Graph Drawing Tools. In *International Symposium on Graph Drawing*. https://doi.org/10.1007/3-540-45848-4_57
- [51] Mary Fernández, Dan Suciu, and Igor Tatarinov. 1999. Declarative Specification of Data-Intensive Web Sites. In *Proceedings of the 2nd Conference on Domain-Specific Languages*. <https://doi.org/10.1145/331960.331979>
- [52] Lisa K Fitzpatrick, Jo Ann Hardacker, Wendy Heirendt, Tracy Agerton, Amy Streicher, Heather Melnyk, Renee Ridzon, Sarah Valway, and Ida Onorato. 2001. A Preventable Outbreak of Tuberculosis Investigated through an Intricate Social Network. *Clinical Infectious Diseases* (2001). <https://doi.org/10.1086/323671>
- [53] Martin Fowler. 2010. *Domain-Specific Languages*. Pearson Education.
- [54] Mark S Fox. 1983. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling*. Technical Report. Carnegie-Mellon University, The Robotics Institute. <https://apps.dtic.mil/docs/citations/ADA138307>
- [55] Ben Frain. 2015. *Responsive Web Design with HTML5 and CSS3*. Packt Publishing.
- [56] Thomas MJ Fruchterman and Edward M Reingold. 1991. Graph Drawing by Force-Directed Placement. *Software: Practice and Experience* (1991). <https://doi.org/10.1002/spe.4380211102>
- [57] Zhuohua Fu, Na He, Song Duan, Qingwu Jiang, Runhua Ye, Yongcheng Pu, Genming Zhao, Z Jennifer Huang, and Frank Y Wong. 2011. HIV Infection, Sexual Behaviors, Sexual Networks, and Drug Use among Rural Residents in Yunnan Province, China. *AIDS and Behavior* (2011). <https://doi.org/10.1007/s10461-010-9797-6>
- [58] Matthew Fuchs. 1997. Domain Specific Languages for ad hoc Distributed Applications. In *Proceedings of the Conference on Domain-Specific Languages*. <https://dl.acm.org/doi/10.5555/1267950.1267953>
- [59] Roy Gal. 2017. Responsive Visualizations coming to Power BI. *Microsoft Power BI Blog* (2017). <https://powerbi.microsoft.com/en-us/blog/responsive-visualizations-coming-to-power-bi/>

- [60] Emden R Gansner, Yehuda Koren, and Stephen North. 2004. Graph Drawing by Stress Majorization. In *International Symposium on Graph Drawing*. https://doi.org/10.1007/978-3-540-31843-9_25
- [61] Nils Gehlenborg, Seán I O'donoghue, Nitin S Baliga, Alexander Goesmann, Matthew A Hibbs, Hiroaki Kitano, Oliver Kohlbacher, Heiko Neuweger, Reinhard Schneider, Dan Tenenbaum, and Anne-Claude Gavin. 2010. Visualization of Omics Data for Systems Biology. *Nature Methods* (2010). <https://doi.org/10.1038/nmeth.1436>
- [62] Burkay Genc and Ugur Dogrusoz. 2003. A Constrained, Force-Directed Layout Algorithm for Biological Pathways. In *International Symposium on Graph Drawing*. https://doi.org/10.1007/978-3-540-24595-7_29
- [63] Helen Gibson, Joe Faith, and Paul Vickers. 2013. A Survey of Two-Dimensional Graph Layout Techniques for Information Visualisation. *Information Visualization* (2013). <https://doi.org/10.1177/1473871612455749>
- [64] Datawrapper GmbH. 2019. Datawrapper. <https://www.datawrapper.de/>. (2019). Accessed: 2020-04-22.
- [65] Pascal Goffin, Jeremy Boy, Wesley Willett, and Petra Isenberg. 2016. An Exploratory Study of Word-Scale Graphics in Data-Rich Text Documents. *IEEE Transactions on Visualization & Computer Graphics* (2016). <https://doi.org/10.1109/TVCG.2016.2618797>
- [66] Pascal Goffin, Wesley Willett, Jean-Daniel Fekete, and Petra Isenberg. 2014. Exploring the Placement and Design of Word-Scale Visualizations. *IEEE Transactions on Visualization & Computer Graphics* (2014). <https://doi.org/10.1109/TVCG.2014.2346435>
- [67] Pascal Goffin, Wesley Willett, Jean-Daniel Fekete, and Petra Isenberg. 2015. Design Considerations for Enhancing Word-Scale Visualizations with Interaction. In *Posters of the Conference on Information Visualization (InfoVis)*. <https://hal.inria.fr/hal-01216216>
- [68] Google. 2020. OpenRefine. <https://openrefine.org/>. (2020). Accessed: 2020-06-23.
- [69] Lars Grammel, Chris Bennett, Melanie Tory, and Margaret-Anne D Storey. 2013. A Survey of Visualization Construction User Interfaces. In *Computer Graphics Forum (Proc. EuroVis, Short Papers)*.

- [70] Scott Grissom, Myles F McNally, and Tom Naps. 2003. Algorithm Visualization in CS Education: Comparing Levels of Student Engagement. In *Proceedings of the ACM Symposium on Software Visualization (SoftVis 2003)*.
<https://doi.org/10.1145/774833.774846>
- [71] Philip J Guo. 2013. Online Python Tutor: Embeddable Web-based Program Visualization for CS Education. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE)*.
<https://doi.org/10.1145/2445196.2445368>
- [72] Philip J Guo, Sean Kandel, Joseph M Hellerstein, and Jeffrey Heer. 2011. Proactive Wrangling: Mixed-Initiative End-User Programming of Data Transformation Scripts. In *ACM User Interface Software & Technology (UIST)*.
<https://doi.org/10.1145/2047196.2047205>
- [73] Matthew Harward, Warwick Irwin, and Neville Churcher. 2010. In Situ Software Visualisation. In *21st Australian Software Engineering Conference (ASWEC)*.
<https://doi.org/10.1109/ASWEC.2010.18>
- [74] Dafang He, Scott Cohen, Brian Price, Daniel Kifer, and C Lee Giles. 2017. Multi-Scale Multi-Task FCN for Semantic Page Segmentation and Table Detection. In *IAPR International Conference on Document Analysis and Recognition (ICDAR)*.
<https://doi.org/10.1109/ICDAR.2017.50>
- [75] Christopher Healey and James Enns. 2012. Attention and Visual Memory in Visualization and Computer Graphics. *IEEE Transactions on Visualization & Computer Graphics* (2012). <https://doi.org/10.1109/TVCG.2011.127>
- [76] Jeffrey Heer. 2018. How Vega Works.
<https://observablehq.com/@vega/how-vega-works>. (2018). Accessed: 2020-02-24.
- [77] Jeffrey Heer, Maneesh Agrawala, and Wesley Willett. 2008. Generalized Selection via Interactive Query Relaxation. In *ACM Human Factors in Computing Systems (CHI)*.
<https://doi.org/10.1145/1357054.1357203>
- [78] Jeffrey Heer and Michael Bostock. 2010. Declarative Language Design for Interactive Visualization. *IEEE Transactions on Visualization & Computer Graphics* (2010).
<https://doi.org/10.1109/TVCG.2010.144>
- [79] Jeffrey Heer, Nicholas Kong, and Maneesh Agrawala. 2009. Sizing the Horizon: The Effects of Chart Size and Layering on the Graphical Perception of Time Series Visualizations. In *ACM Human Factors in Computing Systems (CHI)*.
<https://doi.org/10.1145/1518701.1518897>

- [80] Jeffrey Heer and Ben Shneiderman. 2012. Interactive Dynamics for Visual Analysis. *Queue* (2012). <https://doi.org/10.1145/2133416.2146416>
- [81] Ivan Herman, Guy Melançon, and M Scott Marshall. 2000. Graph Visualization and Navigation in Information Visualization: A Survey. *IEEE Transactions on Visualization & Computer Graphics* (2000). <https://doi.org/10.1109/2945.841119>
- [82] Jefferson Hinke, Isaac Kaplan, Kerim Aydin, George Watters, Robert Olson, and James FK Kitchell. 2004. Visualizing the Food-Web Effects of Fishing for Tunas in the Pacific Ocean. *Ecology and Society* (2004). <https://www.jstor.org/stable/26267649>
- [83] Jane Hoffswell, Alan Borning, and Jeffrey Heer. 2018. SetCoLa: High-Level Constraints for Graph Layout. *Computer Graphics Forum (Proc. EuroVis)* (2018). <https://doi.org/10.1111/cgf.13440>
- [84] Jane Hoffswell, Wilmot Li, and Zhicheng Liu. 2020. Techniques for Flexible Responsive Visualization Design. *ACM Human Factors in Computing Systems (CHI)* (2020). <http://doi.org/10.1145/3313831.3376777>
- [85] Jane Hoffswell and Zhicheng Liu. 2019. Interactive Repair of Tables Extracted from PDF Documents on Mobile Devices. *ACM Human Factors in Computing Systems (CHI)* (2019). <https://doi.org/10.1145/3290605.3300523>
- [86] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2015. Debugging Vega through Inspection of the Data Flow Graph. In *EuroVis Workshop on Reproducibility, Verification, and Validation in Visualization (EuroRV3)*. <http://doi.org/10.2312/eurorv3.20151144>
- [87] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2016. Visual Debugging Techniques for Reactive Data Visualization. *Computer Graphics Forum (Proc. EuroVis)* (2016). <https://doi.org/10.1111/cgf.12903>
- [88] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2018. Augmenting Code with In Situ Visualizations to Aid Program Understanding. *ACM Human Factors in Computing Systems (CHI)* (2018). <https://doi.org/10.1145/3173574.3174106>
- [89] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. <https://doi.org/10.1145/2150976.2151013>

- [90] Tom Horak, Andreas Mathisen, Clemens N Klokmose, Raimund Dachsel, and Niklas Elmqvist. 2019. Vistribute: Distributing Interactive Visualizations in Dynamic Multi-Device Setups. In *ACM Human Factors in Computing Systems (CHI)*. <http://doi.acm.org/10.1145/3290605.3300846>
- [91] Paul Hudak. 1996. Building Domain-Specific Embedded Languages. *ACM Computing Surveys (CSUR)* (1996). <https://doi.org/10.1145/242224.242477>
- [92] Edwin L Hutchins, James D Hollan, and Donald A Norman. 1985. Direct Manipulation Interfaces. *Human-Computer Interaction* (1985). https://doi.org/10.1207/s15327051hci0104_2
- [93] Facebook Inc. 2020. React: A JavaScript Library for Building User Interfaces. <https://reactjs.org/>. (2020). Accessed: 2020-04-29.
- [94] InnateDB. 2014a. Homo Sapiens Gene: DDX58. <http://www.innatedb.com/getGeneCard.do?id=55854>. (2014). Accessed: 2018-03-12.
- [95] InnateDB. 2014b. Homo Sapiens Gene: MAPK1. <http://www.innatedb.com/getGeneCard.do?id=2147>. (2014). Accessed: 2018-03-12.
- [96] InnateDB. 2014c. Homo Sapiens Gene: TLR4. <http://www.innatedb.com/getGeneCard.do?id=82738>. (2014). Accessed: 2018-03-12.
- [97] InnateDB. 2018. NOD-like Receptor Signaling Pathway. <http://www.innatedb.com/interactionSearch.do?from=pw&exPathwayXref=8112&pathwayFilter=&pathwayXrefDB=&pathwayXref=&listType=interaction&coreInteractors=true>. (2018). Accessed: 2018-03-12.
- [98] ITOPF. 2019. Oil Tanker Spill Statistics 2019. <http://www.itopf.org/knowledge-resources/data-statistics/statistics/>. (2019). Accessed: 2020-04-26.
- [99] Eunice Jun, Maureen Daum, Jared Roesch, Sarah Chasins, Emery Berger, Rene Just, and Katharina Reinecke. 2019. Tea: A High-level Language and Runtime System for Automating Statistical Analysis. In *ACM User Interface Software & Technology (UIST)*. <https://doi.org/10.1145/3332165.3347940>

- [100] Sean Kandel, Jeffrey Heer, Catherine Plaisant, Jessie Kennedy, Frank Van Ham, Nathalie Henry Riche, Chris Weaver, Bongshin Lee, Dominique Brodbeck, and Paolo Buono. 2011a. Research Directions in Data Wrangling: Visualizations and Transformations for Usable and Credible Data. *Information Visualization* (2011). <https://doi.org/10.1177/1473871611415994>
- [101] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011b. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *ACM Human Factors in Computing Systems (CHI)*. <https://doi.org/10.1145/1978942.1979444>
- [102] Hyeonsu Kang and Philip J Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *ACM User Interface Software & Technology (UIST)*. <https://doi.org/10.1145/3126594.3126632>
- [103] Kelly A Kearney. 2016. Food Webs as Network Graphs. <http://kellyakearney.net/2016/01/19/food-webs-as-network-graphs-1.html>. (2016). Accessed: 2017-03-14.
- [104] Kelly A Kearney. 2017a. d3-foodweb. <https://github.com/kakearney/d3-foodweb>. (2017). Accessed: 2020-04-25.
- [105] Kelly A Kearney. 2017b. foodwebgraph-pkg. <https://github.com/kakearney/foodwebgraph-pkg>. (2017). Accessed: 2020-04-25.
- [106] Kelly A Kearney, Charles Stock, Kerim Aydin, and Jorge L Sarmiento. 2012. Coupling Planktonic Ecosystem and Fisheries Food Web Models for a Pelagic Ecosystem: Description and Validation for the Subarctic Pacific. *Ecological Modelling* (2012). <https://doi.org/10.1016/j.ecolmodel.2012.04.006>
- [107] Kelly A Kearney, Charles Stock, and Jorge L Sarmiento. 2013. Amplification and Attenuation of Increased Primary Production in a Marine Food Web. *Marine Ecology Progress Series* (2013). <https://doi.org/10.3354/meps10484>
- [108] Curran Kelleher and Haim Levkowitz. 2015. Reactive Data Visualizations. In *Visualization and Data Analysis*. <https://doi.org/10.1117/12.2078301>
- [109] Steve Kieffer, Tim Dwyer, Kim Marriott, and Michael Wybrow. 2013. Incremental Grid-Like Layout Using Soft and Hard Constraints. In *International Symposium on Graph Drawing*. https://doi.org/10.1007/978-3-319-03841-4_39

- [110] Steve Kieffer, Tim Dwyer, Kim Marriott, and Michael Wybrow. 2016. HOLA: Human-Like Orthogonal Network Layout. *IEEE Transactions on Visualization & Computer Graphics* (2016). <https://doi.org/10.1109/TVCG.2015.2467451>
- [111] Nam Wook Kim, Eston Schweickart, Zhicheng Liu, Mira Dontcheva, Wilmot Li, Jovan Popovic, and Hanspeter Pfister. 2016. Data-Driven Guides: Supporting Expressive Design for Information Graphics. *IEEE Transactions on Visualization & Computer Graphics* (2016). <https://doi.org/10.1109/TVCG.2016.2598620>
- [112] Gordon Kindlmann, Charisee Chiw, Nicholas Seltzer, Lamont Samuels, and John Reppy. 2015. Diderot: A Domain-Specific Language for Portable Parallel Scientific Visualization and Image Analysis. *IEEE Transactions on Visualization & Computer Graphics* (2015). <https://doi.org/10.1109/TVCG.2015.2467449>
- [113] Amy J Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. 2011. The State of the Art in End-User Software Engineering. *ACM Computing Surveys (CSUR)* (2011). <https://doi.org/10.1145/1922649.1922658>
- [114] Amy J Ko and Brad A Myers. 2003. Development and evaluation of a model of programming errors. In *IEEE Symposium on Human Centric Computing Languages and Environments*. <https://doi.org/10.1109/HCC.2003.1260196>
- [115] Amy J Ko and Brad A Myers. 2004. Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior. In *ACM Human Factors in Computing Systems (CHI)*. <https://doi.org/10.1145/985692.985712>
- [116] Amy J Ko and Brad A Myers. 2005. A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems. *Journal of Visual Languages & Computing* (2005). <https://doi.org/10.1016/j.jvlc.2004.08.003>
- [117] Amy J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six Learning Barriers in End-User Programming Systems. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. <https://doi.org/10.1109/VLHCC.2004.47>
- [118] Amy J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering* (2006). <https://doi.org/10.1109/TSE.2006.116>

- [119] Stephen G Kobourov. 2012. Spring Embedders and Force Directed Graph Drawing Algorithms. *arXiv preprint arXiv:1201.3011* (2012).
<https://arxiv.org/abs/1201.3011>
- [120] Kaname Kojima, Masao Nagasaki, Euna Jeong, Mitsuru Kato, and Satoru Miyano. 2007. An Efficient Grid Layout Algorithm for Biological Networks Utilizing Various Biological Attributes. *BMC Bioinformatics* (2007).
<https://doi.org/10.1186/1471-2105-8-76>
- [121] Tomaž Kosar, Sašo Gaberc, Jeffrey C Carver, and Marjan Mernik. 2018. Program Comprehension of Domain-Specific and General-Purpose Languages: Replication of a Family of Experiments Using Integrated Development Environments. *Empirical Software Engineering* (2018). <https://doi.org/10.1007/s10664-017-9593-2>
- [122] Tomaž Kosar, Marjan Mernik, and Jeffrey C Carver. 2012. Program Comprehension of Domain-Specific and General-Purpose Languages: Comparison Using a Family of Experiments. *Empirical Software Engineering* (2012).
<https://doi.org/10.1007/s10664-011-9172-x>
- [123] Holger Krahn, Bernhard Rumpe, and Steven Völkel. 2008. Monticore: Modular Development of Textual Domain Specific Languages. In *International Conference on Objects, Components, Models and Patterns*.
https://doi.org/10.1007/978-3-540-69824-1_17
- [124] Kruger National Park 2017. Kruger National Park: What is a Food Web?
<https://kruger-nationalpark.weebly.com/the-food-web.html>. (2017).
Accessed: 2020-04-25.
- [125] Shahid Latif, Diao Liu, and Fabian Beck. 2018. Exploring Interactive Linking Between Text and Visualization. *Computer Graphics Forum (Proc. EuroVis, Short Papers)* (2018). https://www.vis.wiwi.uni-due.de/uploads/tx_itochairt3/publications/091-094.pdf
- [126] David Lavigne. 1996. Cod Food Web.
<http://www.visualcomplexity.com/vc/project.cfm?id=47>. (1996). Accessed: 2017-03-14.
- [127] Weijiang Li and Hiroyuki Kurata. 2005. A Grid Layout Algorithm for Automatic Drawing of Biochemical Networks. *Bioinformatics* (2005).
<https://doi.org/10.1093/bioinformatics/bti290>

- [128] Tom Lieber, Joel R Brandt, and Rob C Miller. 2014. Addressing Misconceptions about Code with Always-On Programming Visualizations. In *ACM Human Factors in Computing Systems (CHI)*. <https://doi.org/10.1145/2556288.2557409>
- [129] Zhicheng Liu, John Thompson, Alan Wilson, Mira Dontcheva, James Delorey, Sam Grigg, Bernard Kerr, and John Stasko. 2018. Data Illustrator: Augmenting Vector Design Tools with Lazy Data Binding for Expressive Visualization Authoring. In *ACM Human Factors in Computing Systems (CHI)*. <https://doi.org/10.1145/3173574.3173697>
- [130] Dastyni Loksa, Benjamin Xie, Harrison Kwik, and Amy J Ko. 2020. Investigating Novices' In Situ Reflections on Their Programming Process. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. <https://doi.org/10.1145/3328778.3366846>
- [131] Kiln Enterprises Ltd. 2020. Flourish. <https://flourish.studio/>. (2020). Accessed: 2020-04-25.
- [132] Jock Mackinlay. 1986. Automating the Design of Graphical Presentations of Relational Information. *ACM Transactions on Graphics* (1986). <https://doi.org/10.1145/22949.22950>
- [133] Ethan Marcotte. 2014. *Responsive Web Design*. A Book Apart Publ.
- [134] Ethan Marcotte. 2015. *Responsive Design: Patterns and Principles*. A Book Apart Publ.
- [135] Michele Mauri, Tommaso Elli, Giorgio Caviglia, Giorgio Uboldi, and Matteo Azzi. 2017. RAWGraphs: A Visualisation Platform to Create Open Outputs. In *Proceedings of the 12th Biannual Conference on Italian SIGCHI Chapter (CHIItaly '17)*. <https://doi.org/10.1145/3125571.3125585>
- [136] D Scott McCrickard and Christa M Chewar. 2003. Attuning Notification Design to User Goals and Attention Costs. *Commun. ACM* (2003). <https://doi.org/10.1145/636772.636800>
- [137] PD McElroy, RB Rothenberg, R Varghese, R Woodruff, GO Minns, SQ Muth, LA Lambert, and R Ridzon. 2003. A Network-Informed Approach to Investigating a Tuberculosis Outbreak: Implications for Enhancing Contact Investigations. *The International Journal of Tuberculosis and Lung Disease* (2003).

- [138] Honghui Mei, Wei Chen, Yuxin Ma, Huihua Guan, and Wanqi Hu. 2018. VisComposer: A Visual Programmable Composition Environment for Enformation Visualization. *Visual Informatics* (2018).
<https://doi.org/10.1016/j.visinf.2018.04.008>
- [139] Gonzalo Gabriel Méndez, Miguel A Nacenta, and Sebastien Vandenheste. 2016. iVoLVER: Interactive Visual Language for Visualization Extraction and Reconstruction. In *ACM Human Factors in Computing Systems (CHI)*.
<https://doi.org/10.1145/2858036.2858435>
- [140] Marjan Mernik, Jan Heering, and Anthony M Sloane. 2005. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys (CSUR)* (2005).
<https://doi.org/10.1145/1118890.1118892>
- [141] Leo A Meyerovich, Arjun Guha, Jacob Baskin, Gregory H Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: A Programming Language for Ajax Applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. <https://doi.org/10.1145/1640089.1640091>
- [142] Microsoft. 2020. Microsoft Excel. <https://products.office.com/en-us/excel>. (2020). Accessed: 2020-04-25.
- [143] Dominik Moritz, Chenglong Wang, Gregory Nelson, Halden Lin, Adam M Smith, Bill Howe, and Jeffrey Heer. 2019. Formalizing Visualization Design Knowledge as Constraints: Actionable and Extensible Models in Draco. *IEEE Transactions on Visualization & Computer Graphics (Proc. InfoVis)* (2019).
<https://doi.org/10.1109/TVCG.2018.2865240>
- [144] Brad A Myers. 1990. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages & Computing* (1990).
[https://doi.org/10.1016/S1045-926X\(05\)80036-9](https://doi.org/10.1016/S1045-926X(05)80036-9)
- [145] Brad A Myers. 1991. Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs. In *ACM User Interface Software & Technology (UIST)*.
<https://doi.org/10.1145/120782.120805>
- [146] Brad A Myers, John F Pane, and Amy Ko. 2004. Natural Programming Languages and Environments. *Commun. ACM* (2004).
<https://doi.org/10.1145/1015864.1015888>

- [147] Jeanne Nakamura and Mihaly Csikszentmihalyi. 2014. The Concept of Flow. In *Flow and the Foundations of Positive Psychology*.
https://doi.org/10.1007/978-94-017-9088-8_16
- [148] Alannah Oleson, Meron Solomon, and Amy J Ko. 2020. Computing Students' Learning Difficulties in HCI Education. In *CHI Conference on Human Factors in Computing Systems*. <https://doi.org/10.1145/3313831.3376149>
- [149] Stephen Oney and Brad Myers. 2009. FireCrystal: Understanding Interactive Behaviors in Dynamic Web Pages. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*.
<https://doi.org/10.1109/VLHCC.2009.5295287>
- [150] John F Pane, Chotirat "Ann" Ratanamahatana, and Brad A Myers. 2001. Studying the Language and Structure in Non-Programmers' Solutions to Programming Problems. *International Journal of Human-Computer Studies* (2001).
<https://doi.org/10.1006/ijhc.2000.0410>
- [151] Chris Parnin and Spencer Rugaber. 2011. Resumption Strategies for Interrupted Programming Tasks. *Software Quality Journal* (2011).
<https://doi.org/10.1007/s11219-010-9104-9>
- [152] Terence Parr. 2009. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf.
- [153] William A Pike, John Stasko, Remco Chang, and Theresa A O'Connell. 2009. The Science of Interaction. *Information Visualization* (2009).
<https://doi.org/10.1057/ivs.2009.22>
- [154] Zening Qu and Jessica Hullman. 2018. Keeping Multiple Views Consistent: Constraints, Validations, and Exceptions in Visualization Authoring. *IEEE Transactions on Visualization & Computer Graphics (Proc. InfoVis)* (2018).
<https://doi.org/10.1109/TVCG.2017.2744198>
- [155] N Quinn and M Breuer. 1979. A Forced Directed Component Placement Procedure for Printed Circuit Boards. *IEEE Transactions on Circuits and Systems* (1979).
<https://doi.org/10.1109/TCS.1979.1084652>
- [156] Jonathan Millard Ragan-Kelley. 2014. *Decoupling Algorithms from the Organization of Computation for High Performance Image Processing*. Ph.D. Dissertation. Massachusetts Institute of Technology. <http://hdl.handle.net/1721.1/89996>

- [157] Casey Reas and Ben Fry. 2006. Processing: Programming for the Media Arts. *AI & SOCIETY* (2006). <https://doi.org/10.1007/s00146-006-0050-9>
- [158] Edward M Reingold and John S Tilford. 1981. Tidier Drawings of Trees. *IEEE Transactions on Software Engineering* (1981). <https://doi.org/10.1109/TSE.1981.234519>
- [159] Donghao Ren, Matthew Brehmer, Bongshin Lee, Tobias Höllerer, and Eun Kyoung Choe. 2017. ChartAccent: Annotation for Data-Driven Storytelling. In *IEEE Pacific Visualization Symposium (PacificVis)*. <https://doi.org/10.1109/PACIFICVIS.2017.8031599>
- [160] Donghao Ren, Tobias Höllerer, and Xiaoru Yuan. 2014. iVisDesigner: Expressive Interactive Design of Information Visualizations. *IEEE Transactions on Visualization & Computer Graphics* (2014). <https://doi.org/10.1109/TVCG.2014.2346291>
- [161] Donghao Ren, Bongshin Lee, and Matthew Brehmer. 2018. Charticulator: Interactive Construction of Bespoke Chart Layouts. *IEEE Transactions on Visualization & Computer Graphics* (2018). <https://doi.org/10.1109/TVCG.2018.2865158>
- [162] Rev.com. 2020. Rev. <https://www.rev.com/>. (2020). Accessed: 2020-05-26.
- [163] Tiark Rompf, Arvind K Sujeeth, HyoukJoong Lee, Kevin J Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. Building-Blocks for Performance Oriented DSLs. *arXiv preprint arXiv:1109.0778* (2011). <http://dx.doi.org/10.4204/EPTCS.66.5>
- [164] Richard B Rothenberg, Claire Sterk, Kathleen E Toomey, John J Potterat, David Johnson, Mark Schrader, and Stefani Hatch. 1998. Using Social Network and Ethnographic Tools to Evaluate Syphilis Transmission. *Sexually Transmitted Diseases* (1998).
- [165] Spencer Rugaber. 2000. The Use of Domain Knowledge in Program Understanding. *Annals of Software Engineering* (2000). <https://doi.org/10.1023/A:1018976708691>
- [166] Peter C de Ruiter, Wolters Volkmar, and John C Moore. 2006. *Dynamic Food Webs: Multispecies Assemblages, Ecosystem Development, and Environmental Change*. <https://doi.org/10.1016/B978-012088458-2/50000-X>
- [167] David Saff and Michael D Ernst. 2003. Reducing Wasted Development Time via Continuous Testing. In *14th International Symposium on Software Reliability Engineering (ISSRE 2003)*. <https://doi.org/10.1109/ISSRE.2003.1251050>

- [168] Cedric Sam. 2018. Ai2html and Its Impact on the News Graphics Industry. *CHI Workshop on Data Visualization on Mobile Devices (MobileVis)* (2018).
https://mobilevis.github.io/assets/mobilevis2018_paper_20.pdf
- [169] Michael John Sannella. 1994. *Constraint Satisfaction and Debugging for Interactive User Interfaces*. Ph.D. Dissertation. University of Washington Seattle, Washington.
- [170] Purvi Saraiya, Chris North, and Karen Duca. 2005. Visualizing Biological Pathways: Requirements Analysis, Systems Evaluation and Research Agenda. *Information Visualization* (2005). <https://doi.org/10.1057/palgrave.ivs.9500102>
- [171] Alper Sarikaya, Michael Correll, Lyn Bartram, Melanie Tory, and Danyel Fisher. 2019. What Do We Talk About When We Talk About Dashboards? *IEEE Transactions on Visualization & Computer Graphics* (2019).
<https://doi.org/10.1109/TVCG.2018.2864903>
- [172] John Sarracino, Odaris Barrios-Arciga, Jasmine Zhu, Noah Marcus, Sorin Lerner, and Ben Wiedermann. 2017. User-Guided Synthesis of Interactive Diagrams. In *ACM Human Factors in Computing Systems (CHI)*.
<https://doi.org/10.1145/3025453.3025467>
- [173] Arvind Satyanarayan and Jeffrey Heer. 2014a. Authoring Narrative Visualizations with Ellipsis. *Computer Graphics Forum (Proc. EuroVis)* (2014).
<https://doi.org/10.1111/cgf.12392>
- [174] Arvind Satyanarayan and Jeffrey Heer. 2014b. Lyra: An Interactive Visualization Design Environment. In *Computer Graphics Forum (Proc. EuroVis)*.
<https://doi.org/10.1111/cgf.12391>
- [175] Arvind Satyanarayan, Bongshin Lee, Donghao Ren, Jeffrey Heer, John Stasko, John R Thompson, Matthew Brehmer, and Zhicheng Liu. 2020. Critical Reflections on Visualization Authoring Systems. *IEEE Transactions on Visualization & Computer Graphics (Proc. InfoVis)* (2020).
<https://doi.org/10.1109/TVCG.2019.2934281>
- [176] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization & Computer Graphics (Proc. InfoVis)* (2017).
<https://doi.org/10.1109/TVCG.2016.2599030>
- [177] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. 2015. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization.

- IEEE Transactions on Visualization & Computer Graphics (Proc. InfoVis)* (2015).
<https://doi.org/10.1109/TVCG.2015.2467091>
- [178] Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. 2014. Declarative Interaction Design for Data Visualization. In *ACM User Interface Software & Technology (UIST)*. <https://doi.org/10.1145/2642918.2647360>
- [179] John Scott. 1988. Social Network Analysis. *Sociology* (1988).
<https://doi.org/10.1177/0038038588022001007>
- [180] Clifford A Shaffer, Matthew Cooper, and Stephen H Edwards. 2007. Algorithm Visualization: A Report on the State of the Field. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE)*.
<https://doi.org/10.1145/1227310.1227366>
- [181] Paul Shannon, Andrew Markiel, Owen Ozier, Nitin S Baliga, Jonathan T Wang, Daniel Ramage, Nada Amin, Benno Schwikowski, and Trey Ideker. 2003. Cytoscape: A Software Environment for Integrated Models of Biomolecular Interaction Networks. *Genome Research* (2003). <http://www.genome.org/cgi/doi/10.1101/gr.1239303>
- [182] Divit P Singh, Lee Lisle, TM Murali, and Kurt Luther. 2018. CrowdLayout: Crowdsourced Design and Evaluation of Biological Network Visualizations. In *ACM Human Factors in Computing Systems (CHI)*.
<https://doi.org/10.1145/3173574.3173806>
- [183] Diomidis Spinellis. 2001. Notable Design Patterns for Domain-Specific Languages. *Journal of Systems and Software* (2001).
[https://doi.org/10.1016/S0164-1212\(00\)00089-3](https://doi.org/10.1016/S0164-1212(00)00089-3)
- [184] Chris Stolte, Diane Tang, and Pat Hanrahan. 2002. Polaris: A System for Query, Analysis, and Visualization of Multidimensional Relational Databases. *IEEE Transactions on Visualization & Computer Graphics* (2002).
<https://doi.org/10.1109/2945.981851>
- [185] M-AD Storey, Kenny Wong, and Hausi A Müller. 2000. How do Program Understanding Tools Affect How Programmers Understand Programs? *Science of Computer Programming* (2000).
[https://doi.org/10.1016/S0167-6423\(99\)00036-2](https://doi.org/10.1016/S0167-6423(99)00036-2)
- [186] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiro Toda. 1981. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man, and Cybernetics* (1981). <https://doi.org/10.1109/TSMC.1981.4308636>

- [187] Amanda Swearngin, Mira Dontcheva, Wilmot Li, Joel Brandt, Morgan Dixon, and Amy J Ko. 2018. Rewire: Interface Design Assistance from Examples. In *ACM Human Factors in Computing Systems (CHI)*. <https://doi.org/10.1145/3173574.3174078>
- [188] Amanda Swearngin, Chenglong Wang, Alannah Oleson, James Fogarty, and Amy J Ko. 2020. Scout: Rapid Exploration of Interface Layout Alternatives through High-Level Design Constraints. *ACM Human Factors in Computing Systems (CHI)* (2020). <https://doi.org/10.1145/3313831.3376593>
- [189] Ben Swift, Andrew Sorensen, Henry Gardner, and John Hosking. 2013. Visual Code Annotations for Cyberphysical Programming. In *Proceedings of the 1st International Workshop on Live Programming*. <https://doi.org/10.1109/LIVE.2013.6617345>
- [190] The New York Times Company. 2017. ai2html. <http://ai2html.org/>. (2017). Accessed: 2020-04-21.
- [191] Trifacta. 2020. Trifacta. <https://www.trifacta.com/>. (2020). Accessed: 2020-06-23.
- [192] Edward R Tufte. 2006. *Beautiful Evidence*. Graphis Pr.
- [193] William Thomas Tutte. 1963. How to Draw a Graph. *Proceedings of the London Mathematical Society* (1963). <https://doi.org/10.1112/plms/s3-13.1.743>
- [194] Arie Van Deursen, Paul Klint, and Joost Visser. 2000. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices* (2000). <https://doi.org/10.1145/352029.352035>
- [195] Jacob VanderPlas, Brian E Granger, Jeffrey Heer, Dominik Moritz, Kanit Wongsuphasawat, Arvind Satyanarayan, Eitan Lees, Ilia Timofeev, Ben Welsh, and Scott Sievert. 2018. Altair: Interactive Statistical Visualizations for Python. *The Journal of Open Source Software* (2018). <https://doi.org/10.21105/joss.01057>
- [196] Vega. 2016. Vega Tutorial. <https://github.com/vega/vega/wiki/Tutorial>. (2016). Accessed: 2020-04-28.
- [197] Bret Victor. 2012a. Inventing on Principle. <https://vimeo.com/36579366>. (2012). Accessed: 2020-04-27.
- [198] Bret Victor. 2012b. Learnable Programming: Designing a Programming System for Understanding Programs. <http://worrydream.com/LearnableProgramming>. (2012). Accessed: 2020-04-25.

- [199] Anneliese von Mayrhauser and A Marie Vans. 1997. Program Understanding Behavior During Debugging of Large Scale Software. In *Workshop on Empirical Studies of Programmers*. <https://doi.org/10.1145/266399.266414>
- [200] Mason Walker. 2019. Americans Favor Mobile Devices over Desktops and Laptops for Getting News. *Pew Research Center* (2019). <https://pewrsr.ch/2uvqS04>
- [201] Zhanyong Wan, Walid Taha, and Paul Hudak. 2002. Event-Driven FRP. In *Practical Aspects of Declarative Languages*. https://doi.org/10.1007/3-540-45587-6_11
- [202] Yunhai Wang, Yanyan Wang, Yinqi Sun, Lifeng Zhu, Kecheng Lu, Chi-Wing Fu, Michael Sedlmair, Oliver Deussen, and Baoquan Chen. 2018a. Revisiting Stress Majorization as a Unified Framework for Interactive Constrained Graph Visualization. *IEEE Transactions on Visualization & Computer Graphics* (2018). <https://doi.org/10.1109/TVCG.2017.2745919>
- [203] Yun Wang, Haidong Zhang, He Huang, Xi Chen, Qiufeng Yin, Zhitao Hou, Dongmei Zhang, Qiong Luo, and Huamin Qu. 2018b. InfoNice: Easy Creation of Information Graphics. In *ACM Human Factors in Computing Systems (CHI)*. <https://doi.org/10.1145/3173574.3173909>
- [204] Hadley Wickham. 2016. *ggplot2: Elegant Graphics for Data Analysis*. Springer.
- [205] Wesley Willett, Jeffrey Heer, and Maneesh Agrawala. 2007. Scented Widgets: Improving Navigation Cues with Embedded Visualizations. *IEEE Transactions on Visualization & Computer Graphics* (2007). <https://doi.org/10.1109/TVCG.2007.70589>
- [206] Kanit Wongsuphasawat, Zening Qu, Dominik Moritz, Riley Chang, Felix Ouk, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. 2017. Voyager 2: Augmenting Visual Analysis with Partial View Specifications. In *ACM Human Factors in Computing Systems (CHI)*. <https://doi.org/10.1145/3025453.3025768>
- [207] Yingcai Wu, Xiaotong Liu, Shixia Liu, and Kwan-Liu Ma. 2012. ViSizer: A Visualization Resizing Framework. *IEEE Transactions on Visualization & Computer Graphics* (2012). <https://doi.org/10.1109/TVCG.2012.114>
- [208] Haijun Xia, Nathalie Henry Riche, Fanny Chevalier, Bruno De Araujo, and Daniel Wigdor. 2018. DataInk: Direct and Creative Data-Oriented Drawing. In *ACM Human Factors in Computing Systems (CHI)*. <https://doi.org/10.1145/3173574.3173797>

- [209] Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. 2020. Penrose: From Mathematical Notation to Beautiful Diagrams. *ACM Transactions on Graphics (TOG)* (2020).
<https://doi.org/10.1145/3386569.3392375>
- [210] Peter Yodzis. 1998. Local Trophodynamics and the Enteraction of Marine Mammals and Fisheries in the Benguela Ecosystem. *Journal of Animal Ecology* (1998).
<https://doi.org/10.1046/j.1365-2656.1998.00224.x>
- [211] Polle T Zellweger, Susan Harkness Regli, Jock D Mackinlay, and Bay-Wei Chang. 2000. The Impact of Fluid Documents on Reading and Browsing: An Observational Study. In *ACM Human Factors in Computing Systems (CHI)*.
<https://doi.org/10.1145/332040.332440>
- [212] Xu Zhu, Miguel Nacenta, Özgür Akgün, and Peter William Nightingale. 2019. How People Visually Represent Discrete Constraint Problems. *IEEE Transactions on Visualization and Computer Graphics* (2019).
<https://doi.org/10.1109/TVCG.2019.2895085>

Appendix A

INTERVIEW RESOURCES: UNDERSTANDING THE BEHAVIOR OF CONSTRAINT SYSTEMS

Section 3.2 describes a set of formative interviews exploring how people use and understand the behavior of constraints. The goal of these interviews was to learn more about existing program understanding practice for constraints and the current limitations encountered when developing constraint-based systems. Prior to the interviews, participants completed a screening survey in which they described their prior experience with constraints and other demographic information. Participants were then contacted by the research team to set up the interview. The questions used in the screening survey are included in Section A.1. The basic interview template is included in Section A.2; the **primary question** is shown in bold, followed by back-up questions that may optionally be used to encourage more discussion.

A.1 Formative Interview Screening Survey

Thank you for your interest in our study on constraint programming systems. The goal of this study is to learn more about how people use, interact with, and understand constraints. This study consists of an informal interview in which you will be asked to share particular anecdotes and impressions regarding your personal experience working with constraints.

If you are interested in participating in this study, please fill out the following survey. This survey aims to learn a bit about your experience to determine if you are a good fit for this study. If you have any questions about this study, please contact <<Interviewer Name>> (<<Interviewer Email>>). If selected to participate in this study, you will be contacted by the research team to schedule the interview. The interview will last about 1 hour, and you will receive a \$20 gift card as compensation.

1. What was the primary goal of the project? How were constraints applied?
2. Why did you decide to use constraints for this project? Please describe any alternative tools or methods that you may have considered.
3. What were the biggest challenges / hurdles you faced in your use of constraints?
4. (Optional) Please share links to any resources you have found particularly useful or to any illustrative examples you would like to share.
5. (Optional) Additional comments

Demographic Information:

6. Name
7. Email
8. Gender
9. Age
10. What is the highest degree or level of school you have completed?
If currently enrolled, what is the highest degree received so far?
11. Current Organizational Affiliation
12. Current Job *For example: "2nd year PhD student," "Associate Professor," "Research Scientist," "Software Developer," etc.*
13. (Optional) Please share any additional comments or information that you think may be relevant to this study.

A.2 Formative Interview Script Template

The goal of these interviews is to learn more about your experience working with constraints. I will ask some questions to get you started, but please feel free to share any information or anecdotes you would like. I would also like to encourage you to share any particular examples or resources that you think might be beneficial to our research team. Before we get started, do you have any questions for me?

1. **Tell me about a particular programming project using constraints that excited or challenged you.** For example, what was the goal of this project? What was the desired end result? What was your background with constraints prior to this project? Can you walk me through your mental model for how the project works?
2. **What was the biggest challenge you faced in working on this project?** What steps did you take to overcome this challenge?
3. **Can you recall a time when the constraints produced a result you did not expect?** What was your process for handling this situation?
4. **At what stage is this project now?** At what point in the process did you feel the project was complete/ready to ship? How did you know that it had reached this point?

[Aside: Discussing and collaborating on constraint-based systems can be particularly difficult, so we aimed to specifically ask about this experience in the interviews.]

5. **Did you collaborate with others during this project?** What was your role on the team?
6. **What parts of the project were most essential to communicate? What parts were the hardest to communicate? What parts could you gloss over or abstract?**

[Aside: For participants that work on end-user facing systems, interesting challenges can arise around how end users interact with the systems. The following questions aim to explore those challenges and any potential interventions the participants have explored.]

7. **What was one of the most surprising applications of your system?**
8. **To the best of your knowledge, what are some of the most common stumbling points expressed by end users of this system?**

[Aside: Next, the interviews aim to encourage a more general discussion of the challenges that arose or are common across multiple projects.]

9. **How many different projects have you worked on involving constraints?**
10. **How common were the challenges you mentioned previously?** During the project you mentioned? Across different projects?

11. **In general, what were the most common hurdles / sticking points that you have encountered when working with constraints?**
12. **Have you ever considered using constraints for a project, but ultimately decided to use a different approach?** What approach did you ultimately use? Why did you decide against using constraints? In what way were they not appropriate or sufficient for the project?

[Aside: Finally, the interviews aim to encourage the participants to think broadly about how their development process might be improved.]

13. **How satisfied are you with your current development process?** How might this process be improved?

Appendix B

HISTORICAL DEBUGGING APPROACH FOR VEGA USING THE JAVASCRIPT CONSOLE

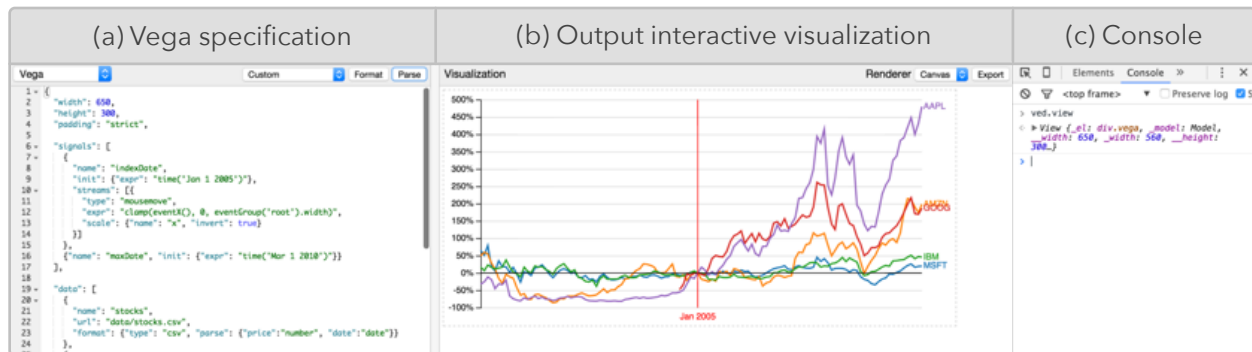


Figure B.1: The historical Vega development and debugging environment. The user writes (a) the Vega specification to produce (b) an output Vega visualization. To debug the behavior, the user can open (c) the JavaScript console of the browser to inspect the underlying system internals.

As described in Section 5.3 of this dissertation, the original Vega development environment did not have debugging support. Instead, users were required to manually traverse the underlying data flow graph and scene graph of Vega’s system internals via the JavaScript console. In order to illustrate this process, consider the following series of steps that would be required to debug the index chart example described in Section 6.2.1. This visualization is an interactive index chart which shows stock prices over time normalized to the location of an interactive cursor. As the user moves the cursor, the lines in the stock chart update based on the date-time that corresponds to the cursor location. The former Vega IDE consists of a user-defined specification (Figure B.1a), an output visualization (Figure B.1b), and access to the system internals via the JavaScript console (Figure B.1c).

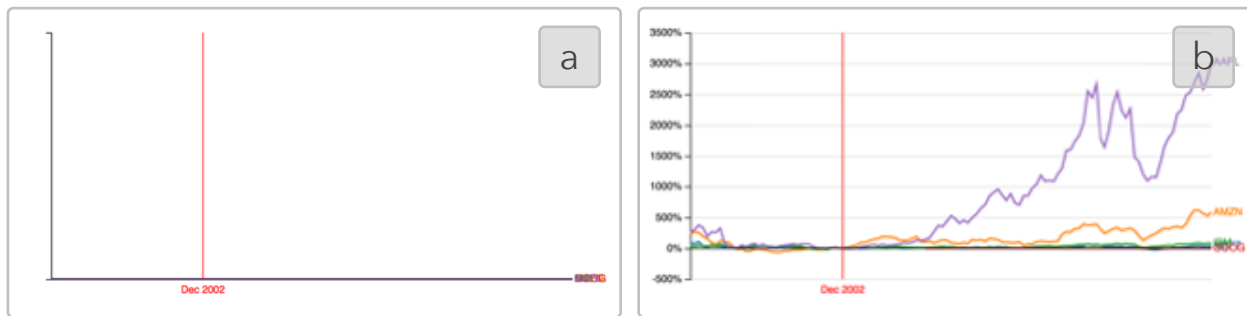


Figure B.2: (a) The broken state for an interactive index chart, as compared to (b) the correct version of the visualization at the same time point.

Step 1: Identify the problem state. Upon observing an error (Figure B.2a), the user must isolate the problem state. In the index chart, the error occurs at a particular timestamp for the interactive cursor, which is only a few pixels wide and thus difficult to hit exactly.

Step 2: View the system internals. The user must know that the system internals can be accessed from the JavaScript console via the command shown here: `ved.view` (Figure B.3a).

Step 3: What should the user look for? The user notices that no lines appear to be drawn in this error case (Figure B.2a). In response, the user wants to navigate the Vega scene graph to identify what marks appear on the canvas (if any). The Vega scene graph can be accessed via the command: `ved.view.model().scene()` (Figure B.3b).

Step 4: Navigating the scene graph. By navigating the scene graph, the user can locate the group that corresponds to the set of line marks (Figure B.3c) and locate the part of the scene graph corresponding to the individual lines (Figure B.3d). The user may therefore wonder: since the lines appear to exist, why are they not visible?

Step 5: Inspect the encoding. The line mark is defined by multiple points with an x and y value. Based on a random sampling of the values, it seems that while the x value varies the y value is always the same: 265 (Figure B.3e).

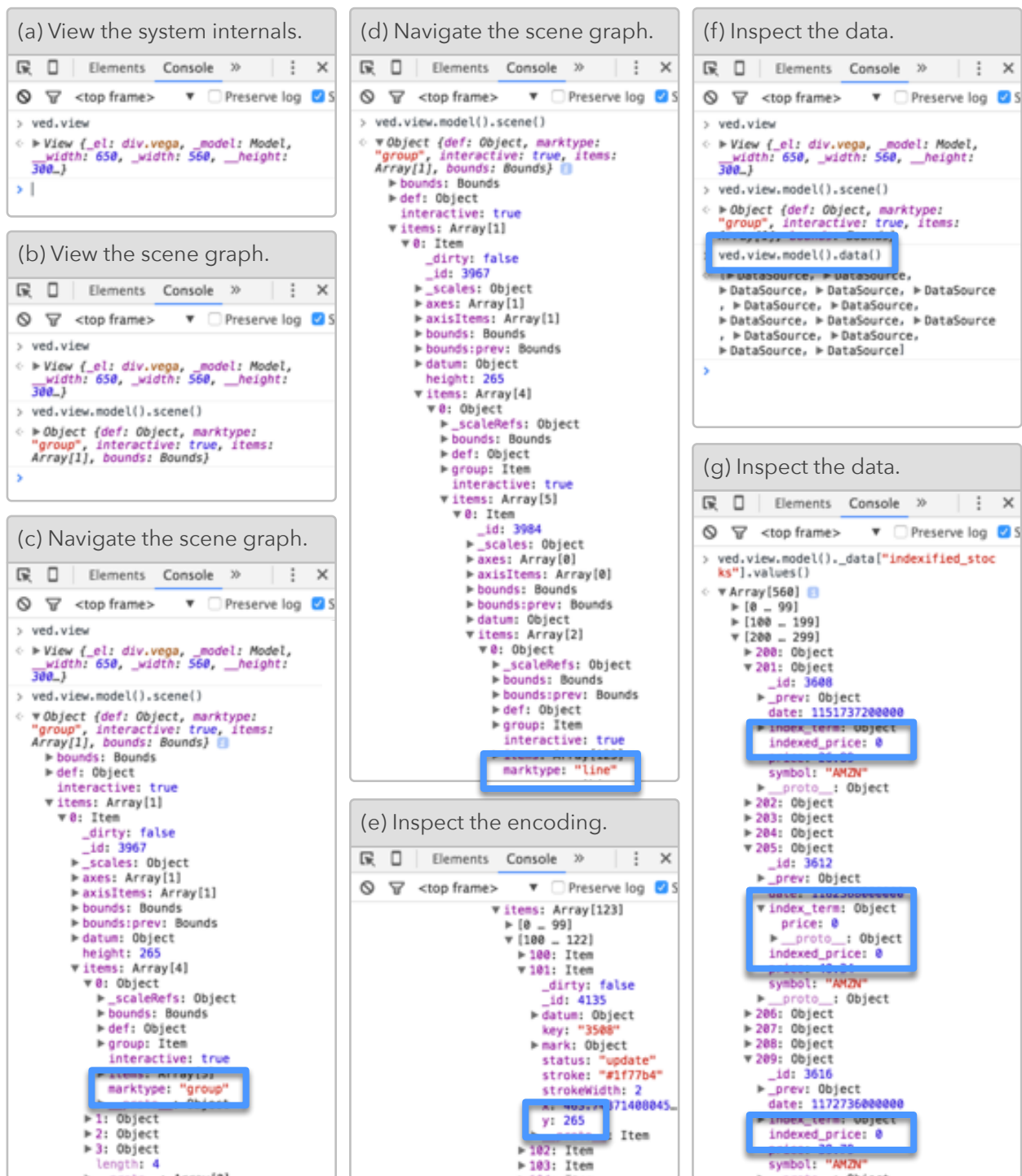


Figure B.3: Snapshots of different points in the debugging process described in this section. Each of these screenshots captures a particular state of the JavaScript console. To really understand the behavior, users must be able to effectively navigate this complex internal structure.

(a) The Vega code responsible for creating the line marks.

```

75 -   {
76     "type": "group",
77 -   "from": {
78     "data": "indexified_stocks",
79     "transform": [{"type": "facet", "groupby": ["symbol"]}]}
80   },
81 -   "marks": [
82     {
83       "type": "line",
84       "properties": {
85         "update": {
86           "x": {"scale": "x", "field": "date"},
87           "y": {"scale": "y", "field": "indexed_price"},
88           "stroke": {"scale": "color", "field": "symbol"},
89           "strokeWidth": {"value": 2}
90         }
91       }
92     },

```

(b) The Vega code responsible for creating the backing "indexified_stocks" dataset.

```

34 -   {
35     "name": "indexified_stocks",
36     "source": "stocks",
37 -   "transform": [{
38     "type": "lookup",
39     "on": "index", "onKey": "symbol",
40     "keys": ["symbol"], "as": ["index_term"],
41     "default": {"price": 0}
42 -   }, {
43     "type": "formula",
44     "field": "indexed_price",
45     "expr": "datum.index_term.price > 0 ? (datum.price - datum.index_term.pri
46   }]}
47 }

```

Figure B.4: (a) The Vega code responsible for creating the line marks. (b) The code for the `indexified_stocks` dataset which defines the `indexed_price` variable used in the line encoding.

Step 6: How is the y value determined? Inspecting the visualization, the user notices that the lines are not missing, they are flat against the y axis. In particular, the user notices that the text is visible at the end of the line (Figure B.2a). By locating the point in the specification where the y value is defined, the user can identify that this value is drawn from a data field: `indexed_price` (Figure B.4a).

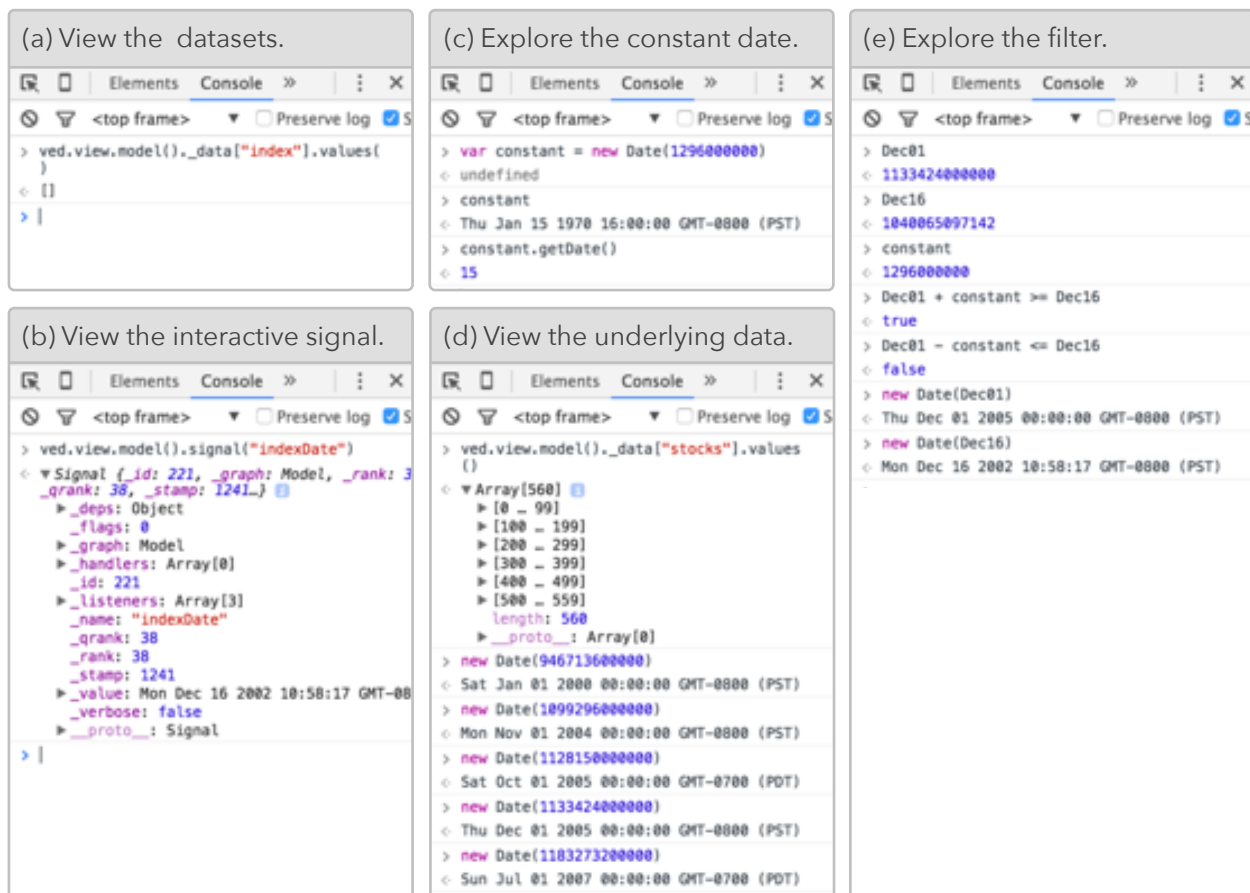


Figure B.5: Snapshots of different points in the debugging process described in this section. Each of these screenshots captures a particular state of the JavaScript console.

Step 7: How is indexed_price defined? The `indexed_price` variable is a field in the `indexed_stocks` dataset, so the user can locate that part of the code and find the calculation for `indexed_price`. However, this line is so long that it does not fit in the editor window (Figure B.4b, Line 45). Looking at this calculation, the user can figure out that if the `index_term` is not greater than zero, then `indexed_price` is set to zero. The user may therefore wonder: what is the `index_term`?

Step 8: Inspect the data. The user can return to the console to locate the data in the console via the command: `ved.view.model().data()` (Figure B.3f). A sampling of data for the `indexed_stocks` dataset shows that for all points, both the `indexed_price` and `index_term` are zero (Figure B.3g). The user may wonder: why is `index_term` always zero?

(a) The Vega code responsible for creating the "index" dataset.

```

26 ▾ {
27   "name": "index",
28   "source": "stocks",
29 ▾   "transform": [{
30     "type": "filter",
31     "test": "datum.date + 1296000000 >= indexDate && datum.date - 1296000000 <= i
32   }]
33 },

```

(b) The Vega code responsible for specifying the behavior of the interactive cursor.

```

7 ▾   "signals": [
8 ▾     {
9       "name": "indexDate",
10      "init": {"expr": "time('Jan 1 2005')"},
11 ▾     "streams": [{
12       "type": "mousemove",
13       "expr": "clamp(eventX(), 0, eventGroup('root').width)",
14       "scale": {"name": "x", "invert": true}
15     }]
16     },
17     {"name": "maxDate", "init": {"expr": "time('Mar 1 2010')}}
18   ],

```

Figure B.6: The Vega code for defining (a) the `index` dataset and (b) the interactive cursor.

Step 9: Why is `index_term` zero? Looking at the code, the `index_term` is defined as a lookup on the `index` dataset with a default of zero (Figure B.4b). Then, what is `index`?

Step 10: What does `index` look like? Returning to the console, the user inspect the values of the `index` dataset to examine why the lookup is using the default or why the value identified from the lookup is zero (Figure B.5a). However, the user finds that there are no values in the dataset, which raises the question: how is the `index` dataset defined?

Step 11: How is `index` defined? Looking at the specification, the user can find the definition of the `index` dataset (Figure B.6a). The user can see that `index` is a filter on `stocks` based on a complex `test` definition. Based on Step 10, it looks like the filter removes everything. But why is this the case? The filter tests each data point in `stocks` (the `datum`) plus or minus some value (1296000000) against `indexDate`. So then what is `indexDate`?

Step 12: What is `indexDate`? `indexDate` is defined as a signal that updates on mousemove to get the current time value (Figure B.6b). Using the console, we can find the value of the signal for `indexDate` to see if there is a problem in the interaction. We can extract the value using the command: `ved.view.model().signal("indexDate")` (Figure B.5b). However, looking at the result, it seems that the value (Dec. 16, 2002) is reasonable. The filter tests each data point in `stocks` (`datum`) plus or minus some value against `indexDate`. Revisiting the definition of the `index` dataset, we find that the filter tests each data point in `stocks` plus or minus some value against `indexDate`. So then how does this equation work? The value is kept if: `date - value <= indexDate <= date + value`.

But to understand how this equation works, the user must first understand the behavior of the constant (1296000000). The constant specifies the range in which the data tuples pass the filter, so what is the constant doing? The constant looks like unix time, similar to other values in the data set. We can use the console to explore what it might mean (Figure B.5c): the date corresponds to Jan. 15th, which does not really make sense for a generic filter. The day is the 15th. The `indexDate` was Dec. 16th, so we are looking for all data values such that Dec. 16th falls within the range: `date` plus or minus 15.

Step 13: What are the dates in the dataset? To understand how values work with this range, we can use the console to inspect the `stocks` dataset to identify what dates are being tested in the filter (Figure B.5d). From a random sampling of points in the dataset, the user can see that every point corresponds to the first of the month. So for each data point, what happens with the filter? If we look at Dec. 01, we see that it does not pass the filter. And neither do any of the other dates (Figure B.5e). In particular, notice that there is a time associated with the index point as well as the date, such that the `indexDate` is equal to “Mon Dec 16 2002 10:58:17 GMT-0800 (PST)” (Figure B.5b). However, the data in the `stocks` dataset does not have an associated time (Figure B.5d). Two of these dates are compared side-by-side in Figure B.5e: in this case, `Dec01` corresponds to data from the `stocks` dataset and `Dec16` corresponds to the `indexDate`.


```

26 ▾ {
27   "name": "index",
28   "source": "stocks",
29 ▾   "transform": [{
30     "type": "filter",
31     "test": "month(datum.date) == month(indexDate) && year(datum.date) == year(i
32   }]
33 },

```

Figure B.7: The Vega code to fix the broken index chart uses the month and year to appropriately filter the data, rather than an usual calculation involving a constant (Figure B.6a). The visualization will no longer flatline (Figure B.2a) and instead will produce the desired result (Figure B.2b).

Step 14: The answer. No values pass the filter because the associated time is pushing them out of the range. The filter is attempting to capture the point associated with the month and year of the index point. In particular, the constant is trying to create a month-wide range around the current date. But this does not work for all index dates and times. By updating the equation for the filter to factor in the actual month calculation correctly (Figure B.7), the visualization starts to work as expected (Figure B.2b).

Appendix C

INTERVIEW RESOURCES: VISUALIZING VEGA'S BEHAVIOR AS A DATA FLOW GRAPH

Section 5.4 describes a set of formative interviews exploring how expert Vega users understand and debug the output Vega visualizations. The goal of these interviews was to learn more about existing program understanding practice and the current challenges around debugging in Vega. The interviews also aimed to explore the potential utility of a prototype data flow graph visualization for Vega (see Section 5.3). The basic interview template is included in Section C.1. The example data flow graph visualizations shown to participants during the interview are included in Section C.2.

C.1 Formative Interview Script Template

The goal of these interviews is to gather insight into your development and debugging process.

1. Walk me through the process you use when working with Vega.
2. What version of Vega are you currently using?
3. What is your primary method for producing Vega specifications? To clarify, methods include: (1) by hand, (2) modifications to existing specifications, (3) programmatically generated by your own code, (4) programmatically generated by another system.
4. What was the last (or most troublesome) error you encountered when generating a Vega specification?
 - (a) What was your approach towards resolving this problem?
 - (b) What information was most important in identifying the problem?
 - (c) What tools did you use to facilitate the debugging process?
5. In what ways do you think the debugging process could have been facilitated?

6. Are there any other errors that you found particularly difficult to debug?
 Repeat Question 5 and Question 6 if so.

Up to this point, we have mainly discussed your current strategies when developing and debugging Vega. I would now like to dive a little deeper into what strategies or techniques could be added to facilitate the development process.

7. In general, what additional functionality or information could be added to facilitate your development process?
8. One debugging strategy is to use the console to inspect the underlying execution structure and scene graph of Vega. How often do you use the console for debugging Vega?
- (a) To what extent do you examine the underlying execution structure of Vega?
 - (b) Do you think this structure is something you could/will use in the future for debugging Vega specifications? If so, how? If not, why not?

The JavaScript console allows the user to access a lot of potentially useful information, but may require more domain knowledge or an intricate understanding of the Vega internals. One way to reduce the difficulty associated with inspecting the textual structure is to provide a visualization of the underlying structure. This example shows the spec, visualization, and underlying data flow graph (Section C.2).

9. What is your initial impression of the data flow graph? In particular, what do you notice? What are you confused by?
10. Do you think that this structure would be useful for the development process?
- (a) **(yes)** What parts of this representation do you think would be most useful? What sorts of interactions or information do you think would be relevant or helpful?
 - (b) **(no)** Why do you think this representation would not be useful? Is there any information within this structure that would be useful to surface in another way?
11. What do you think could be added or changed to improve this visual representation?

Those are all the questions I have. Thank you for taking the time to discuss these problems and topics with me. Please feel free to reach out if you have any additional thoughts on the topic or uncover other debugging scenarios that could be facilitated.

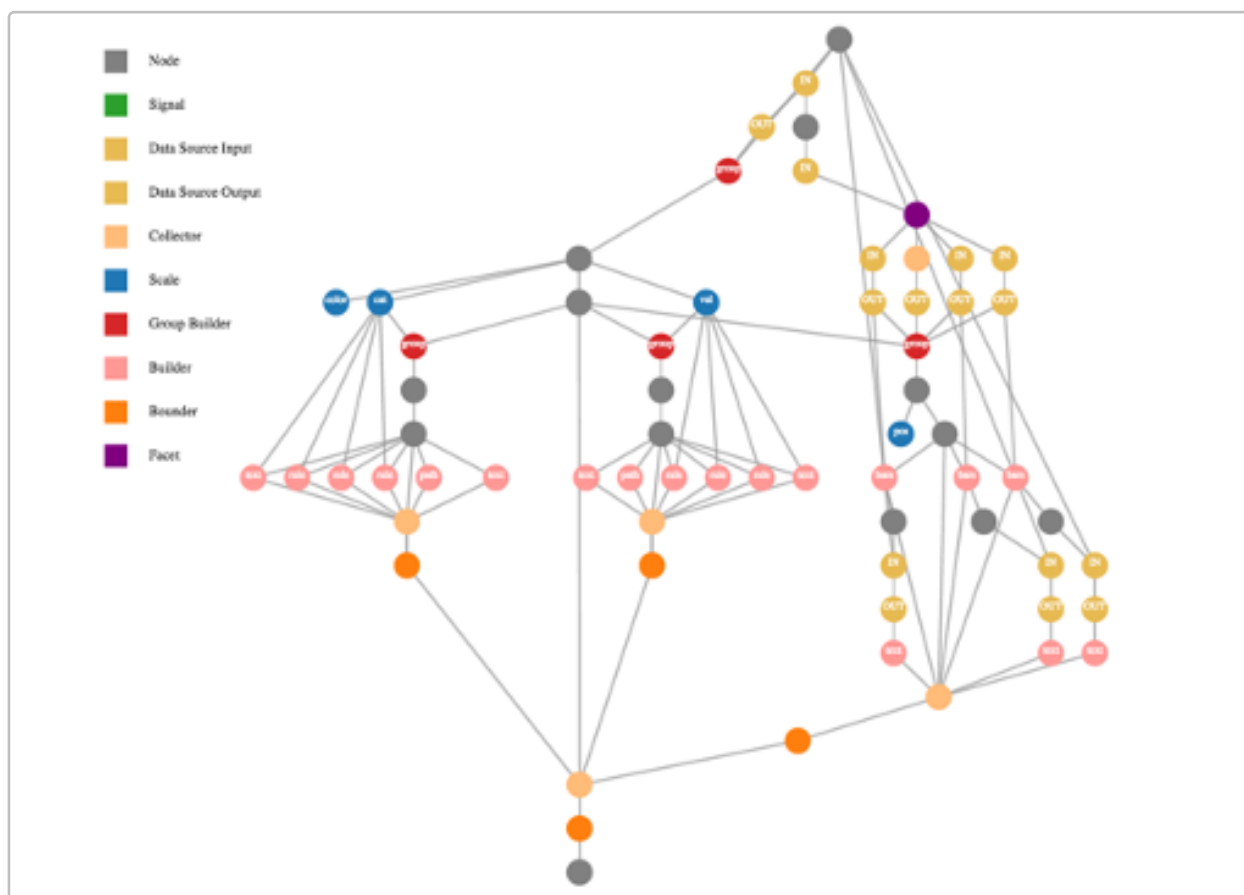


Figure C.1: A data flow graph of a grouped bar chart showing the axis marks (“cat” and “val”). Note: only the figure was shown during the interview; this caption was added to explain the content.

C.2 Data Flow Example Visualizations

For the formative interviews on program understanding techniques for Vega, we provided several example visualizations of the Vega data flow graph (see Section 5.3). In this section, we include the four resources that were shown to participants during the interviews. These resources show the data flow graph with varying degrees of detail and annotation. Figure C.1 shows the full, unannotated data flow graph for a grouped bar chart (Figure 5.2a). Figure C.2 shows the same data flow graph with annotations, the corresponding Vega specification, and output visualization. Figure C.3 shows a simplified version of the data flow graph alongside the Vega code and output. Finally, Figure C.4 shows the same information as Figure C.3, but with the code highlighted corresponding to the data flow graph.

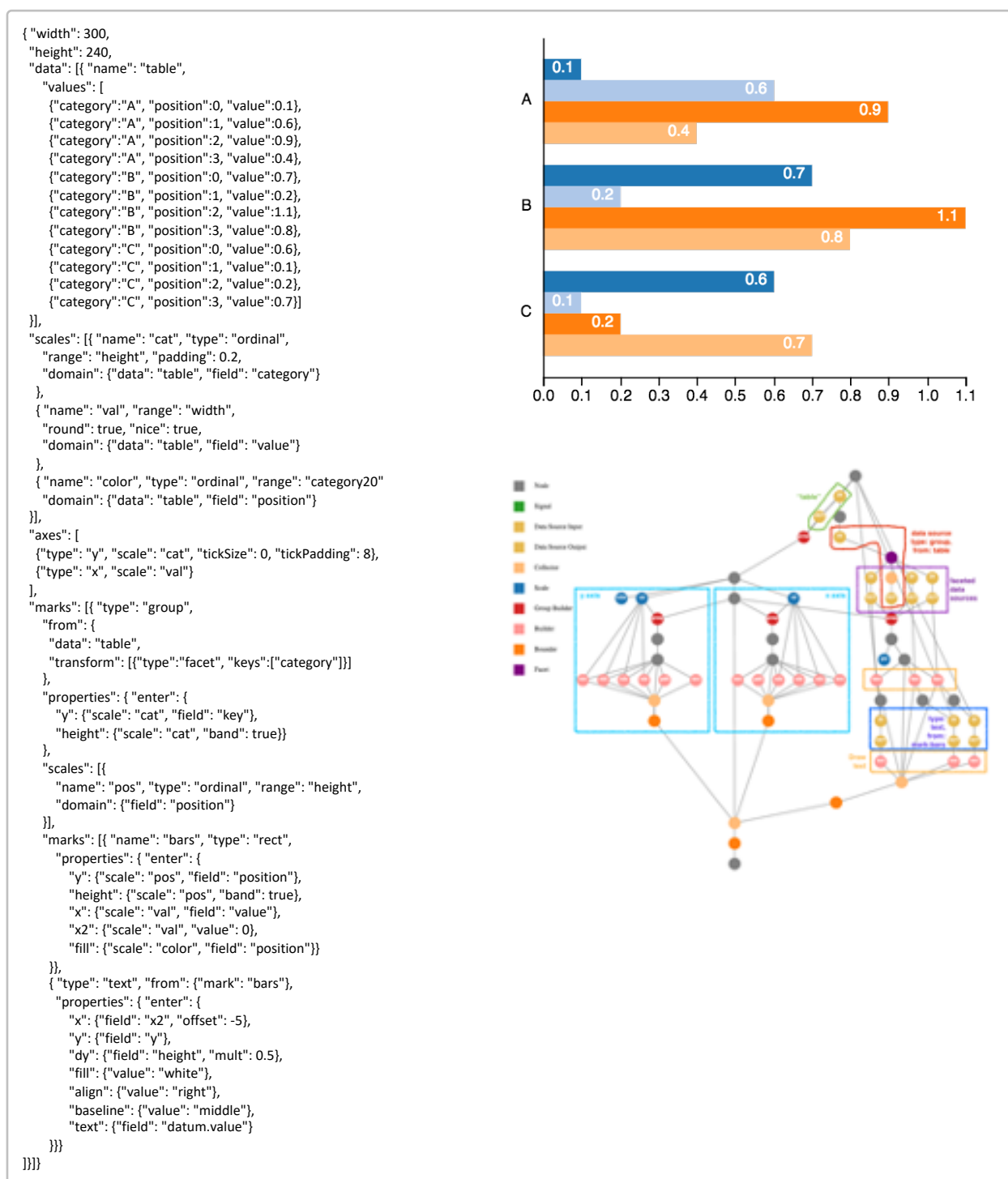


Figure C.2: The Vega specification, output visualization, and annotated data flow graph for a grouped bar chart visualization. The annotations show which parts of the graph correspond to relevant Vega components such as the axes and data. Note: only the figure was included in the original interview materials; this caption was added here to explain the content and rationale.

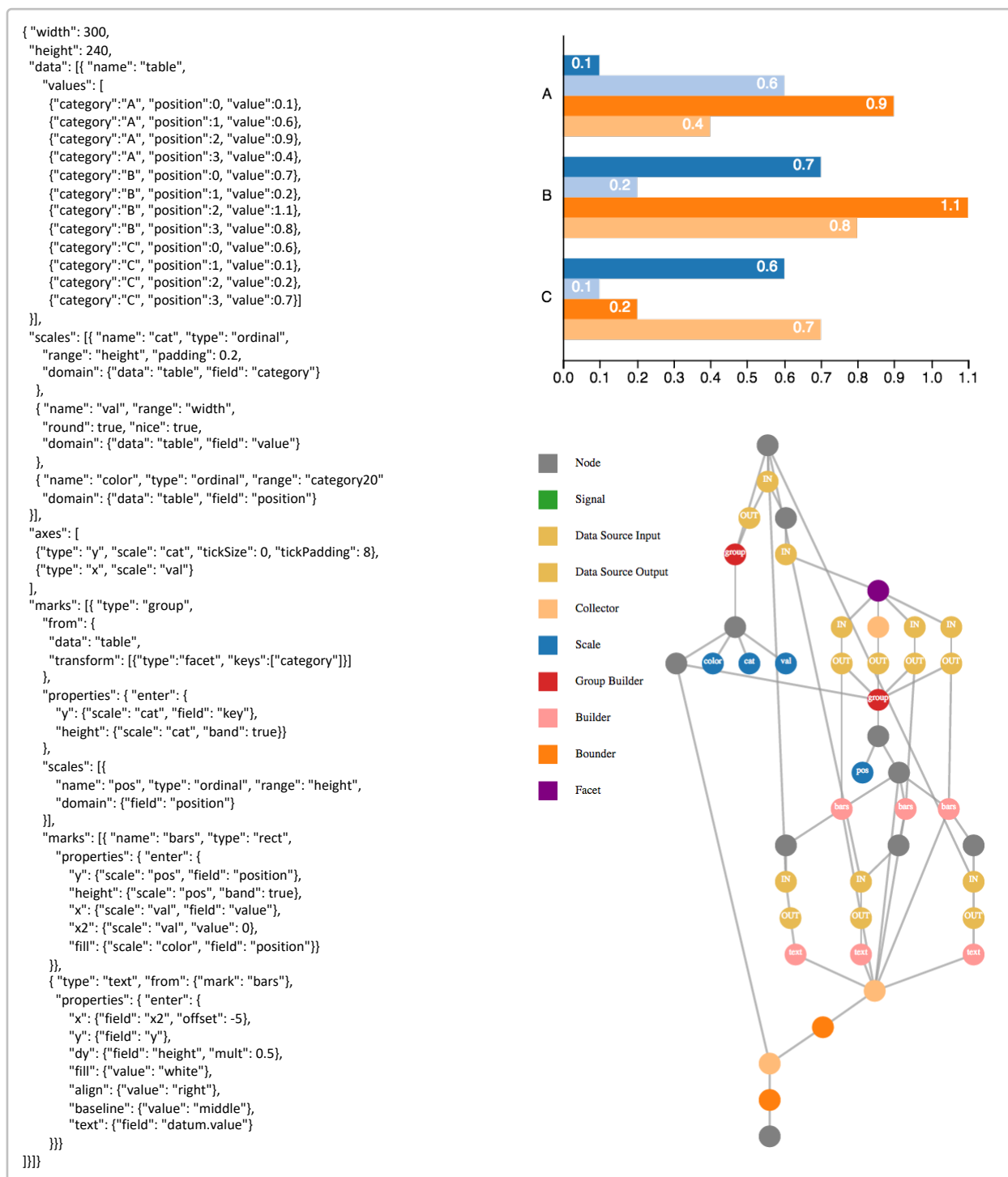


Figure C.3: The Vega specification, output visualization, and simplified data flow graph for a grouped bar chart visualization. The data flow graph has been simplified to hide the nodes corresponding to the axes, which are visible in Figure C.1. Note: only the figure was included in the original interview materials; this caption was added here to explain the content and rationale.

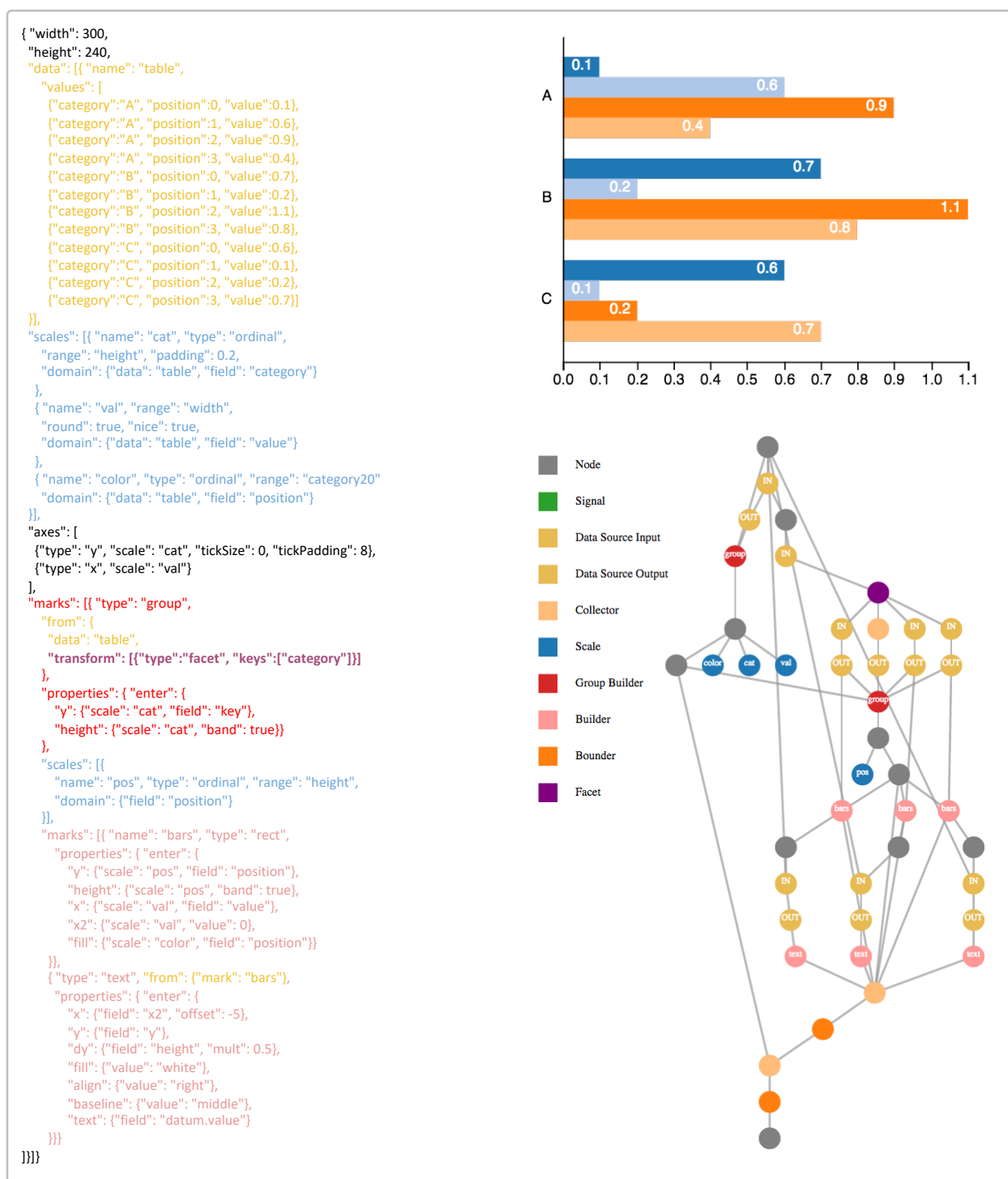


Figure C.4: The Vega specification, output visualization, and simplified data flow graph for a grouped bar chart. The code is highlighted to match the corresponding elements in the data flow graph visualization, and thus help visualize the relationship between the two components. Note: only the figure was shown during the interview; this caption was added here to explain the content.

Appendix D

EVALUATION RESOURCES: VISUAL DEBUGGING TECHNIQUES FOR VEGA

Section 6.2 describes our evaluation of the visual debugging techniques for Vega. This appendix includes the resources used to conduct the evaluation including: (1) the evaluation survey questions (Section D.1) and (2) the evaluation reference sheet (Section D.2).

D.1 Evaluation Post-Task Survey and Exit Survey

Participants completed three evaluation tasks: (1) debugging the data transformations on an interactive index chart, (2) debugging panning on a scatterplot, and (3) debugging brushing on a scatterplot. The task protocol is described in more detail in Section 6.2. Participants completed a post-task survey (Section D.1.2) after each task to identify candidate lines from the specification as the source of the error and to rate the visual debugging techniques. At the end of the evaluation, participants completed a set of exit survey questions (Section D.1.3).

D.1.1 Introductory Information: Debugging Vega

For this study you will use a new debugging environment to identify errors in three interactive Vega visualizations. You will first be given an introduction to the debugging environment and a Vega specification during which time you are free to ask any questions. For each of the debugging tasks, you will be told what the desired functionality should be. You will then:

1. Test the interactions and identify the bug.
2. Identify one (or more) lines of the specification that you believe contributes to the bug.
3. Explain your reasoning.
4. Rate the debugging strategies available.

Click continue to see the questions for the first task.

D.1.2 Post-Task Survey Questions

1. Identify a line number of the (unmodified) specification that contributes to the bug. *This can be your best guess answer.*
2. Explain the reasoning for your answer above. *To help answer this question, consider answering some of the following questions: (1) what is the bug, (2) why do you think the bug occurs, (3) how did you reach this conclusion?*
3. How useful was the debugging environment for identifying the source of the bug? *On a scale from 1 (“Distracting”) to 5 (“Essential”) for the nine debugging environment features: overview, replay, timeline, dependencies, tooltip, annotations, data table, change summary, and specification (see Appendix D.2).*
4. In what ways, if any, could the debugging environment have better helped you identify the source of the bug?

D.1.3 Exit Survey Questions

1. Rank the usefulness of each feature in developing interactive visualizations in Vega. *On a scale from 1 (“Unhelpful”) to 8 (“Essential”) for the eight debugging environment features: overview, replay, timeline, dependencies, tooltip, annotations, data table, and change summary (see Appendix D.2).*
2. What feature would you remove, if any? *Why would you remove this feature?*
3. What feature would you like to add, if any? *Please explain your addition.*
4. *(Optional)* Please share any additional impressions or comments.

Demographic Information: Remember, all of the demographic information gathered here will be reported anonymously so will not be used to identify you.

5. Name
6. Email: *What email would you like the amazon gift card sent to?*
7. Age
8. Gender

9. Department
10. Level: *year and/or position (e.g., 2nd year PhD Student)*
11. Please describe your familiarity with Vega, if any, prior to this user study.
12. (Optional) Please share any additional information that you think may be relevant to this study, if any.

D.2 Evaluation Reference Sheet

For the user evaluation described in Section 6.2, participants were provided with a reference sheet of the visual debugging techniques available in Vega. Participants also received a brief tutorial prior to the evaluation task. The reference sheet is included in Figure D.1.

Replay Indicator:

- record: interact with the vis. which updates timeline
- replay: display past state and tooltip

Dependencies:

highlights the signal & value on which used to define the signal at mouseover (dark gray)

Change Summary: (only with data tab)

the line charts show how the distribution of values in each property (id, _id) changes over time.

id	_id
42	338
132	339
118	340
110	352

Overview:

bar height shows # signals changed in pulse

Timeline:

squares show user defined signals over time

Specification:

user defined code for visualization

Arrow keys move cursor through timeline.

or click the square of interest.

Data Table:

selecting the data tab shows the data table.

id	selectedPoints	sepal_length	sepalWidth	petalLength	petalWidth	species	_id	_prev
5.1	3.5	1.4	0.2	'setosa'	1	['_id':1,'sepalLength':5.1,'...		
4.9	3	1.4	0.2	'setosa'	2	['_id':2,'sepalLength':4.9,'...		
4.7	3.2	1.3	0.2	'setosa'	3	['_id':3,'sepalLength':4.7,'...		
4.6	3.1	1.5	0.2	'setosa'	4	['_id':4,'sepalLength':4.6,'...		
5	3.6	1.4	0.2	'setosa'	5	['_id':5,'sepalLength':5,'...		
5.4	3.9	1.7	0.4	'setosa'	6	['_id':6,'sepalLength':5.4,'...		
4.8	3.4	1.4	0.3	'setosa'	7	['_id':7,'sepalLength':4.8,'...		
5	3.4	1.5	0.2	'setosa'	8	['_id':8,'sepalLength':5,'...		
4.4	2.9	1.4	0.2	'setosa'	9	['_id':9,'sepalLength':4.4,'...		
4.9	3.1	1.5	0.1	'setosa'	10	['_id':10,'sepalLength':4.9,'...		

histograms show the distribution of value in at the current time.

Cursor:

the current point in the timeline. the most recent value for a given signal is shown in light green and the value is displayed in the rightmost column.

Annotations:

Signals corresponding to a point are drawn on the visualization.

The current point is bordered in white and colored based on: dark (past) > red (current) > white (future)

Tooltip: (only during replay)

information about the underlying visualization state

Change Summary: (only with data tab)

the line charts show how the distribution of values in each property (id, _id) changes over time.

Dependencies:

highlights the signal & value on which used to define the signal at mouseover (dark gray)

Replay Indicator:

record: interact with the vis. which updates timeline

replay: display past state and tooltip

Figure D.1: A reference sheet describing the visual debugging techniques and interactions available in Vega. The reference sheet could be used at any point during the user evaluation (see Section 6.2).

Appendix E

EVALUATION RESOURCES: AUGMENTING CODE WITH IN SITU VISUALIZATIONS

Section 7.3 describes our evaluation of the code augmentations for program understanding in Vega. This appendix includes the resources used to conduct the evaluation including: (1) the screening survey for potential participants (Section E.1); (2) the evaluation scripts for both conditions (Section E.2); (3) the instruction sheet for the evaluation (Section E.3); (4) the instruction sheet introducing the in situ visualizations (Section E.4); (5) the program understanding questions for the training task (Section E.5); (6) the program understanding questions for the evaluation tasks (Section E.6); and (7) the exit survey (Section E.7).

E.1 Evaluation Screening Survey

Are you curious about the interplay between code and program output? Are you interested in how new program understanding tools can make it easier to learn a new programming language? If so, we would like to invite you to participate in a research study on how program visualizations in code can influence program understanding. We are looking for individuals to complete a 90 minute session on the University of Washington campus between Sept. 11th and Sept. 15th. In this study, you will be asked to review two programs and answer program understanding questions about the behavior of the code. You will receive a \$20 Amazon gift card as a thank you for your participation. If you are interested in participating in this study, please complete the following survey questions to determine if you are eligible. This screening survey should take about 5 minutes. To participate in the study, you must be at least 18 years of age and must have prior programming experience. Participants will be excluded from participating in this study if they have previously participated in a Vega study on debugging

interactive visualizations. Participation in this study is voluntary and any identifying information gathered during this study will be kept confidential. If you have any questions about this study, please contact <<Interviewer Name>> by email at <<Interviewer Email>> or by phone at <<Interviewer Phone Number>>.

Eligibility Information:

1. Name
2. Email: *At what email can we contact you to finish scheduling this study?*
3. Are you at least 18 years of age, or older?
4. Please indicate whether you have had programming experience from the following sources: computer science classes, personal programming projects, software engineering internships, software engineering jobs, research internships, full-time research positions, other. **Indicate yes/no for each potential source.**
5. Please briefly describe your prior programming experience.
6. How frequently do you use the following programming languages: Python, Java, JavaScript, C or C++, Lisp or Racket, Ruby, SQL, Vega, other. **Ratings provided for each language as one of: never, rarely, occasionally, regularly.**
7. If you answered “Other” above, what other programming languages are you most familiar with?
8. Please describe your prior experience with Vega, if any. *Vega is a declarative language for creating, saving, and sharing interactive visualization designs. Please note that no prior experience with Vega is required for this study.*
9. Please select all times during which you would be available to participate in the study. *If eligible, you will be contacted to finalize a time slot for the study. Please note that the study is expected to take about 90 minutes. [Aside: Respondents were asked to fill out a day/time grid of potential study session slots.]*
10. Would you like to be contacted if additional time slots are added outside this window?
11. **(Optional)** Please include any comments or questions you have regarding this screening survey or the research study. *You are also welcome to email <<Interviewer Name>> (<<Interviewer Email>>) with any questions.*

E.2 Evaluation Script

Participants completed two tasks, one in each of two conditions: *baseline* and *visualization*. We counterbalanced the conditions across participants, such that some participants started with the *baseline* condition whereas other participants started with the *visualization* condition. In this section, we included the two scripts that were used depending on which condition participants were placed in first. Only one script was used per participant and outlines the behavior for both of the evaluation tasks (e.g., the entire evaluation period). Notice that the scripts generally include the same content, but in a different order based on the condition.

E.2.1 Evaluation Script for the Baseline Condition

Thank you for agreeing to participate in this study on visualizing program behavior. Participation is voluntary and you may leave at any time. Before we start, I would like to check if it is okay for me to record the screen/audio for this study session? Please take a chance now to look over this consent form.

[Aside: Turn on audio/screen recording. Collect signed consent form.]

For this study, you will be presented with a training task and two study tasks in which you will answer program understanding questions about Vega programs. You are welcome to ask any questions while reviewing the instructions and completing the training task, however, I will be unable to answer any questions once the study tasks have begun. I want to make it clear right away that we're evaluating the system, not you. Please answer all questions to the best of your ability. Feel free to think aloud or explain your answers in writing as much as you would like for any question. Here is a piece of scratch paper you may use for taking any notes you would like during the tasks. At this point, please take a chance to review this instruction sheet for an introduction to Vega.

[Aside: Give the participant the instruction sheet (see Section E.3).]

Now I will give you a set of training questions in our system so you can familiarize yourself with the task setup. On the left is the Vega code and the right is the output Vega visualization. The bottom right shows the task instruction and questions. You may interact with the Vega visualization at any point while answering the task questions. We encourage you to thoroughly explore all the possible states or settings of the visualization. Once you have reviewed the instructions, feel free to begin the training at any time.

[Aside: Let the participant complete the training task (see Section E.5).]

Here is the first task. Feel free to ask any questions before you begin and remember that I cannot answer any questions once you start the task. Please review the instructions and begin whenever you're ready.

[Aside: Let the participant complete the first task.]

For the second task, the Vega code will be annotated with code visualizations of the program behavior. Here is a brief introduction to the code visualizations. Please take a moment to review this information; you may once again experiment with the demo task to see the code visualizations at work. Feel free to ask any questions at this time and let me know when you are ready to continue to the second task.

[Aside: Give the participant the instruction sheet (see Section E.4) and demo.]

Here is the second task; once you have reviewed the instructions, feel free to begin the task.

[Aside: Let the participant complete the second task.]

You have now completed all the tasks. Here is a final survey about your experience with the code visualizations and environment.

[Aside: Give the participant the exit survey (see Section E.7).]

E.2.2 Evaluation Script for the Visualization Condition

Thank you for agreeing to participate in this study on visualizing program behavior. Participation is voluntary and you may leave at any time. Before we start, I would like to check if it is okay for me to record the screen/audio for this study session? Please take a chance now to look over this consent form.

[Aside: Turn on audio/screen recording. Collect signed consent form.]

For this study, you will be presented with a training task and two study tasks in which you will answer program understanding questions about Vega programs. You are welcome to ask any questions while reviewing the instructions and completing the training task, however, I will be unable to answer any questions once the study tasks have begun. I want to make it clear right away that we're evaluating the system, not you. Please answer all questions to the best of your ability. Feel free to think aloud or explain your answers in writing as much as you would like for any question. Here is a piece of scratch paper you may use for taking any notes you would like during the tasks. At this point, please take a chance to review this instruction sheet for an introduction to Vega.

[Aside: Give the participant the instruction sheet (see Section E.3).]

Now I will give you a set of training questions in our system so you can familiarize yourself with the task setup. On the left is the Vega code and the right is the output Vega visualization. The bottom right shows the task instruction and questions. You may interact with the Vega visualization at any point while answering the task questions. We encourage you to thoroughly explore all the possible states or settings of the visualization. Once you have reviewed the instructions, feel free to begin the training at any time.

[Aside: Let the participant complete the training task (see Section E.5).]

For the first task, the Vega code will be annotated with code visualizations of the program behavior. Here is a brief introduction to the code visualizations. Please take a moment to

review this information; you may once again experiment with the demo task to see the code visualizations at work. Feel free to ask any questions at this time and let me know when you are ready to continue to the first task.

[Aside: Give the participant the instruction sheet (see Section E.4) and demo.]

Here is the first task. Feel free to ask any questions before you begin and remember that I cannot answer any questions once you start. Please review the instructions and begin whenever you're ready.

[Aside: Let the participant complete the first task.]

For the second task, the Vega code will not include the code visualizations you just saw. Once you have reviewed the instructions, feel free to begin at any time.

[Aside: Let the participant complete the second task.]

You have now completed all the tasks. Here is a final survey about your experience with the code visualizations and environment.

[Aside: Give the participant the exit survey (see Section E.7).]

E.3 Evaluation Instruction Sheet

For this study, you will be shown a set of visualizations written in the language Vega. A Vega specification defines an interactive visualization in a JSON format. Vega provides basic building blocks for a wide variety of visualization designs: **data** loading and transformation, **scales**, **axes**, and graphical **marks** such as rectangles, lines, plotting symbols, etc. Interaction techniques can be specified using reactive **signals** that dynamically modify a visualization in response to input event streams. To create a visualization in Vega, you start with a dataset like the one shown in Figure E.1c. You can then write a Vega specification that defines **scales**, **axes**, and **marks** based on the data. The following specification (Figure E.1a) creates a bar chart visualization (Figure E.1b).

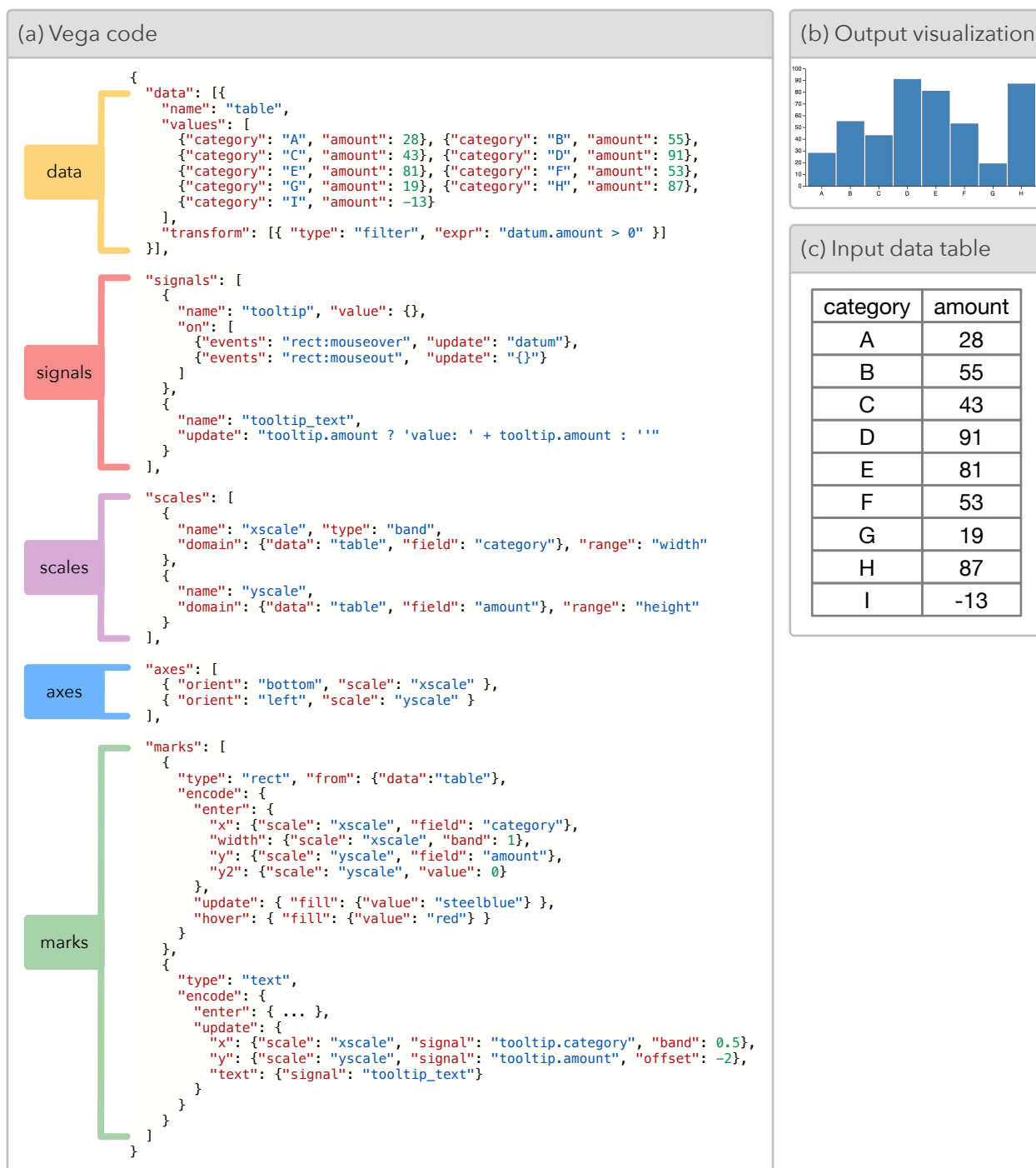


Figure E.1: The Vega example used for training in the evaluation of the code augmentations includes (a) the annotated Vega code for (b) a bar chart visualization based on (c) a small set of input data. This example illustrates the relevant Vega visualization components described in the training and the behavior of data transformations. Note: only the figure was included in the original evaluation materials; this caption was added here to explain the content and rationale.

A Vega specification can have multiple **datasets** defined under “data.” Each dataset is a set of values that can have multiple data fields, which are referenced by a named “field.” In this example there is one dataset (“table”), which has two data fields (“category” and “amount”). Dataset definitions can also define **transforms** that can modify the values in the dataset. For example, this visualization includes a “filter” transformation to only include values (“datum”) from the dataset that have an “amount” greater than zero.

Scales are functions that can map data values (e.g., “category” name) to visual properties of the marks (e.g., “x” position). You can define **axes** to visualize the scales. This Vega specification includes two **axes**, one for the category label on the “bottom” and one to indicate the height of each bar on the “left.”

Marks define the visual encoding for the visualization. This Vega visualization defines a bar chart by creating a “rect” mark for each element in the dataset. Visual properties of the “rect” mark can be parameterized by data fields (identified as a named “field”) from the dataset. For example, the “x” position of the bar is parameterized by the “category” field of the “table” dataset to determine the placement of the bar within the chart. For this encoding, the field value is first passed through the corresponding scale (e.g., “xscale”).

You can introduce interactivity into a visualization by defining “signals” that are dynamic variables that are updated by interaction events on the visualization and can drive interactive updates of the visual encoding. For example, the following signal definition can be added to the specification to capture hover events on the bars. **Signals** can either update *directly* in response to interaction events (e.g., mouseover or click) or can experience downstream changes when updating in response to other signals.

Here are a few notable features of the code environment and important keywords. Please feel free to refer to this fact sheet at any point during the tasks for useful reminders.



Figure E.2: We augmented the online Vega editor to display a tooltip showing the signal value or summary information about the values of a data field (e.g., the min, max, and mean). The tooltip reflects the general behavior found in existing debugging environments [9]. Note: only the figure was included in the original evaluation materials; this caption was added to explain the content.

Keywords:

Specification: The Vega code in JSON format.

Signals: Dynamic variables derived from interaction events.

Data field: The values within a dataset identified by the column (“field”) name.

Mark: A graphical element of the visualization (e.g., rectangles, lines, text, etc.).

Visual property: The appearance of a mark (e.g., position, size, color, etc.).

Visual encoding: The mapping from data fields to the visual properties of marks in the visualization (e.g., amount \rightarrow y position).

Direct updates: Updates to a signal, dataset, or visual property that occur directly in response to an interaction event.

Downstream changes: Updates to a signal, data, or visual property that occur in response to other updates to the specification (but not directly in response to an interaction event).

Code Environment:

Search: You can search the Vega code with command-F while the code pane is live.

Parse: You can reset the visualization to the default state by clicking “Parse.”

Tooltips: You can view the current value of a signal by hovering over the signal name. You can view information about data fields (e.g., the min/max/mean) by hovering over the data field name as shown in Figure E.2.

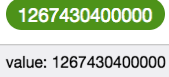
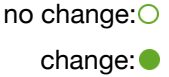

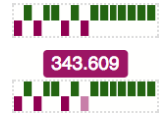
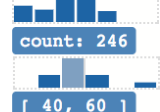
	value	The exact value of the signal at the current point or summary information about the data field.
	indicator	Whether or not the signal value has changed between the previous and current state of the visualization. <i>Interaction: Mousing over the indicator shows the value or change.</i>
	line	The signal value for the 15 previous time steps. <i>Interaction: Mousing over the line shows the value of the signal at that point. Holding shift while doing so shows the value for all signals.</i>
	tick	Whether or not the signal value has changed for the 15 previous time steps (green: change, magenta: no change). <i>Interaction: Mousing over a tick shows the value of the signal at that point. Holding shift while doing so shows the value for all signals.</i>
	histogram	A histogram showing the distribution of values for that data field. The bin ranges are optimized for the current state of the visualization and may change while interacting with the Vega visualization. <i>Interaction: Mousing over a bar shows the bin range and the count.</i>

Figure E.3: For the user evaluation, we selected a subset of in situ visualizations to include (see Section 7.2). To help evaluation participants interpret these (potentially new) visualization designs, we included the descriptions seen in this figure. Note: only the figure was included in the original evaluation materials; this caption was added here to explain the content and rationale.

E.4 Evaluation Instruction Sheet for the In Situ Visualizations

Code visualizations appear for two types of Vega tokens (**signals** and **data fields**) while interacting with the output. Figure E.3 includes a brief description of the code visualizations you may see. Mousing over the code visualization provides extra information such as the value of the signal (line/tick) or the number of values for that range in the dataset (histogram).

E.5 Evaluation Training Tasks

This Vega program creates a basic bar chart. Mousing over the bars shows a tooltip with the bar value. For this task you will be asked a series of questions about the behavior of the code. Please answer the questions as quickly and completely as possible. Once you have submitted an answer, you will not be able to change it; if at some point you feel that one of your previous answers was wrong, please provide your new answer along with an explanation in the text box for the current question *in addition* to your answer for the current question.

You are free to interact with the Vega visualization and code visualizations as much as you would like for each question, but you cannot change the code itself. You may collapse parts of the code using the buttons in the margin. You can reset the visualization to the initial state by clicking “Parse.” You are welcome to ask any questions about the task or programming environment at this point. You may continue to ask questions during the training questions. However, we cannot answer any questions once the study task has begun.

Press “Start” when you are ready to begin the task. Please take this opportunity to thoroughly explore all possible states or settings of the visualization. When you feel you have sufficiently explored the visualization and are ready to continue to the next step, press “Submit.”

1. What is the name of the primary dataset being visualized?
2. What are the visual encodings of the visualization (e.g., the mapping from data fields to visual properties of the marks)? *For each answer, please include the name of the data field, the dataset it came from, and the visual field it encodes (e.g., the “field name” field from the “example” dataset encodes the point “size”).*
3. What is the primary mark type of the visualization?
4. Which signals update while interacting with the visualization?
If no signals update, respond “None.”
5. Which signals update in response to mouseover events directly?
Your response may overlap with your previous answer.
6. Which datasets update while interacting with the visualization?
If no datasets update, respond “None.”
7. Which datasets never update during interaction?

E.6 Task-Specific Program Understanding Questions

During the evaluation, participants repeated the same set of questions once for the *baseline* and once for the *visualization* condition, each time with a different Vega visualization task. There were four possible visualization tasks (though each participant only completed two of

them during the study session). The full methodology is described in Section 7.3. For task specific questions, we inserted the following text to customize the template.

<u>Task</u>	<u>Description</u>	<u>Interaction</u>
Population	“a population chart showing the number of individuals in each age bracket for both women and men. Dragging the slider shows data for different years.”	“slider”
Index	“an index chart that shows stock information. The lines for each stock are normalized relative to the interactive cursor.”	“mousemove”
Scatterplot	“a scatterplot of points. You can pan the visible region by dragging the background.”	“drag”
Overview	“an overview+detail visualization. Selecting a region in the smaller visualization zooms the larger region to show that range in more detail. You may also drag the selected region.”	“drag”

E.6.1 Program Understanding Question Template

This Vega program creates `<<task description>>`.

Please answer the questions as quickly and completely as possible. Once you have submitted an answer, you will not be able to change it; if at some point you feel that one of your previous answers was wrong, please provide your new answer along with an explanation in the text box for the current question **in addition** to your answer for the current question.

You are free to interact with the Vega visualization and code visualizations as much as you would like for each question, but you cannot change the code itself. You may collapse parts of the code using the buttons in the margin. You can reset the visualization to the initial state by clicking “Parse.” You are welcome to ask any questions about the task or programming environment now. However, we cannot answer any questions once the study task has begun.

Press “Start” when you are ready to begin this task. Please take this opportunity to thoroughly explore all possible states or settings of the visualization. When you are ready to continue to the next step, press “Submit.”

1. What is the name of the primary dataset being visualized?
2. What are the visual encodings of the visualization (e.g., the mapping from data fields to visual properties of the marks)? *For each answer, please include the name of the data field, the dataset it came from, and the visual field it encodes (e.g., the “field name” field from the “example” dataset encodes the point “size”).*
3. What is the primary mark type of the visualization?
4. For the mark type identified in Q3, how many marks are created for this visualization? *If you would like to change your previous answer, please write your new answer and an explanation in the text box **in addition** to your answer to the current question.*
5. Which signals update while interacting with the visualization? *If no signals update, respond “None.”*
6. Which signals update in response to <<task interaction>> events directly? *Your response may overlap with your previous answer.*
7. Which signals never update during interaction?
8. For each signal identified in Q5, how is it used throughout the code? *If you would like to change your previous answer, please write your new answer and an explanation in the text box **in addition** to your answer to the current question.*
9. Which signal is used most frequently to parameterize the data transformations and visual encodings?
10. For the signal identified in Q9, what is the range of values (e.g., min and max) for that signal? *If you would like to change your previous answer, please write your new answer and an explanation in the text box **in addition** to your answer to the current question.*
11. Which datasets update while interacting with the visualization? *If no datasets update, respond “None.”*
12. Which datasets update directly in response to the signals from Q5? *Your response may overlap with your previous answer. If you would like to change your previous answer, please write your new answer and an explanation in the text box **in addition** to your answer to the current question.*
13. Which datasets never update during interaction?

14. For each dataset identified in Q11, what data fields are used to parameterize the code? *For each response, please include the dataset name, field name, and how the data field is used in your answer. If you would like to change your previous answer, please write your new answer and an explanation in the text box **in addition** to your answer to the current question.*
15. Which data field is used most frequently throughout the code to parameterize the visual encodings? *Please also include the dataset from which this field is taken.*
16. For the data field identified in Q15, what is the range of values (e.g., min and max value) for that field? *If you would like to change your previous answer, please write your new answer and an explanation in the text box **in addition** to your answer to the current question.*
17. Which data field exhibits the largest changes during interaction? *Please also include the dataset from which this field is taken. If no data fields exhibit changes during interaction, respond “None.”*
18. Does the visualization exhibit any unexpected behaviors during interaction?
 - (a) [If no, submit the form.](#)
 - (b) [If yes, continue to following questions.](#)
19. Please describe the unexpected behavior exhibited by the visualization.
20. How did you become aware of this unexpected behavior?
21. Which signals and or data fields give rise to this unexpected behavior? *Please explain your answer and include the dataset from which the data field is taken.*
22. Which signals and or data fields exhibit or are impacted by this unexpected behavior? *Please explain your answer and include the dataset from which the data field is taken.*

E.7 Evaluation Exit Survey

You have now completed all the tasks for this study. Please complete the following survey about your experience with the programming environment and code visualizations.

1. How quickly were you able to answer the task questions? [On a scale from 1 \(“Quicker with unedited code”\) to 7 \(“Quicker with code visualizations”\).](#)

2. How accurate were you at answering the task questions? On a scale from 1 (“More accurate with unedited code”) to 7 (“More accurate with code visualizations”).
3. (Optional) Please share any other thoughts about your experience with the programming environment and code visualizations.

Code Visualizations:

During this study you saw a couple different types of code visualizations: line, tick, and histogram. Please describe your experience with the various code visualizations.

4. How helpful was each visualization type for completing the tasks? On a scale from 1 (“Not helpful”) to 5 (“Extremely helpful”), for each of the five visualizations.
5. How interpretable was each visualization type for completing the tasks? On a scale from 1 (“Not interpretable”) to 5 (“Extremely interpretable”), for each of the five visualization types.
6. How intrusive was each visualization type while completing the tasks? On a scale from 1 (“Not intrusive”) to 5 (“Extremely intrusive”), for each of the five visualization types.
7. (Optional) Please share any other thoughts about your experience with the code visualizations.

Demographic Information:

8. Name
9. Email: *What email would you like your gift card sent to?*
10. Age
11. Gender
12. Organizational Affiliation: *If your answer is the UW, please include your departmental affiliation as well.*
13. Position: *School year and/or position (e.g., 2nd year PhD student, faculty, etc.)*
14. (Optional) Please share any additional information or comments that you think may be relevant to this study.

Appendix F

INTERVIEW RESOURCES: RESPONSIVE VISUALIZATION DESIGN PRACTICES

For the formative interviews about journalists' responsive visualization design practices (see Section 8.2), we used a common interview question template (Section F.1), which was customized prior to each interview to ask specific questions about past articles that the journalist had worked on (Section F.1.1). These customizations aimed to highlight particular responsive techniques that might not otherwise arise during conversation. During the interviews themselves, we asked additional follow-up questions based on the flow of conversation. The basic interview template is included in the following section; customizations are shown in [blue](#). After the interviews, we reviewed the conversations to extract common themes and challenges described by the journalists, the results of which are described in Section 8.2.

F.1 Formative Interview Script Template

Thank you for agreeing to talk to me. As I mentioned in my email, we are interested in your development process for responsive visualizations that adapt to different devices.

1. To start out, how often and in what ways do you collaborate with others while working on an article and developing the visualizations?
2. Can you walk me through your general workflow when developing a visualization?
3. What tools or techniques are you using to produce the visualization? What tools are you using for prototyping, data exploration, or producing the original design?
4. Can you describe how you iterate on visualization designs? Do you often develop alternative designs and how do you decide between them?

5. For our earlier analysis, we were looking at your article <<article title>> from <<organization>>. Can you walk me through the visualizations in this article and describe your design rationale with respect to the responsive techniques used?
 - a. <<see article specific questions below>>
6. We were also looking at your article <<article title>>. Can you walk me through some of the visualizations in this article and describe your rationale?
 - a. <<see article specific questions below>>
7. When working on visualizations in general, how often and why do you decide *not* to make the visualization responsive?
8. Which device do you prioritize or design for in your usual workflow (if any)?
9. What is your process for producing a responsive visualization? How does the design of responsive visualizations fit into your general workflow and at what point in the process do you consider the responsive aspects of the design?
10. Are you designing multiple versions or producing a single adaptive version?
11. How often do you consider changing what data is included in the mobile alternative (for example, by filtering or aggregating the underlying data)?
12. Can you think of any other examples of projects you've worked on in which you've used interesting responsive techniques that you might want to share with us?
13. Thinking more broadly, how important do you think it is for visualizations to be responsive to different devices? In particular, how important do you think it is to produce radically different versions for different devices?
14. What are the major challenges that you've faced when trying to design responsive visualizations?
15. How might your workflow be improved to support the development of more responsive visualization designs?
16. At this point, those are all of the questions I had prepared. Do you have any questions for me or final thoughts about responsive visualization design?

F.1.1 Customized Article Specific Questions

For each interview, we also came up with some questions specific to the article that we wanted to discuss; these questions aimed to thoroughly explore some of the responsive design decisions in the article. Here are some examples of the questions we asked.

1. How did you decide which annotations to include on the chart at various points?
2. Why did you decide to remove these annotations from the mobile version?
3. How did you decide on which details to remove and which to keep?
4. How did you come up with the original design and how did you decide what simplifications to make?
5. Can you discuss the different interaction techniques exhibited by the desktop and mobile versions?
6. For the visualization <<visualization title>>, you use somewhat different binning schemes between the desktop and mobile versions. How did you decide on the bin ranges and what impact do you think that has on this visualization?

Appendix G

RESPONSIVE VISUALIZATION CORPUS BIBLIOGRAPHY

To identify the types of responsive visualization techniques used in online news articles, we conducted a survey of 53 articles from twelve news organizations. To produce this corpus, we surveyed best-of lists for visualization designs and the visualization galleries for particular sources of interest. For more information on our analysis, procedure, and results, see Section 8.3.1. The list of selected articles for this corpus is included here:

- [1] Gregor Aisch. 2014. The Clubs That Connect the World Cup. *The New York Times* (2014). <https://nyti.ms/2nCtGXG>
- [2] Gregor Aisch and Karen Yourish. 2015. Connecting the Dots Behind the 2016 Presidential Candidates. *The New York Times* (2015). <https://nyti.ms/2ykNIYP>
- [3] Susanne Barton and Hannah Recht. 2018. The Massive Prize Luring Miners to the Stars. *Bloomberg* (2018). <https://www.bloomberg.com/graphics/2018-asteroid-mining/>
- [4] David Batty, Caelainn Barr, and Pamela Duncan. 2018. What Lies Beneath: The Subterranean Secrets of London’s Super-Rich. *The Guardian* (2018). <https://www.theguardian.com/money/2018/may/07/going-underground-the-subterranean-secrets-of-londons-super-rich>
- [5] Gurman Bhatia. 2018. India’s Premium Price of Petrol. *Reuters Graphics* (2018). <https://fingfx.thomsonreuters.com/gfx/rngs/INDIA-ELECTION-FUEL/010080DM0SB/index.html>
- [6] Gurman Bhatia and Manas Sharma. 2019. The Figures Behind the Faces. *Reuters Graphics* (2019). <https://graphics.reuters.com/INDIA-ELECTION-CRIMINAL-CANDIDATES/0100925031T/index.html>
- [7] Seth Blanchard and Reuben Fischer-Baum. 2018. One of the World Cup’s Best Goals was Even Crazier Than You Thought. *The Washington Post* (2018). https://wapo.st/2Nu5yk9?tid=ss_tw

- [8] Jay Boice and Rachael Dottle. 2018. 2018 March Madness Predictions. *FiveThirtyEight* (2018). <https://projects.fivethirtyeight.com/2018-march-madness-predictions/>
- [9] Larry Buchanan and Karen Yourish. 2018. From Criminal Convictions to Ethical Lapses: The Range of Misconduct in Trump's Orbit. *The New York Times* (2018). <https://nyti.ms/2N3WGVj>
- [10] Weiyi Cai. 2018. War of Words. *Reuters Graphics* (2018). <http://fingfx.thomsonreuters.com/gfx/rngs/NORTHKOREA-USA-KIM-TRUMP/010070JM16P/index.html>
- [11] Manuel Canales and Sean McNaughton. 2019. See Which Countries Fund the Most Scientific Research. *National Geographic* (2019). <https://www.nationalgeographic.com/magazine/2019/05/data-show-why-china-science-research-is-booming/>
- [12] Pia Catton and Gus Wezerek. 2018. Nearly Half The Kentucky Derby Field Is Racing Against A Half-Brother. *FiveThirtyEight* (2018). <https://53eig.ht/2H0JSQH>
- [13] Sahil Chinoy. 2018. The Places in the U.S. Where Disaster Strikes Again and Again. *The New York Times* (2018). <https://nyti.ms/2GJjoe4>
- [14] Sahil Chinoy and Jessica Ma. 2019. How Every Member Got to Congress. *The New York Times* (2019). <https://www.nytimes.com/interactive/2019/01/26/opinion/sunday/paths-to-congress.html>
- [15] Matt Daniels. 2019. The Largest Vocabulary In Hip Hop. *The Pudding* (2019). <https://pudding.cool/projects/vocabulary/>
- [16] Fathom Information Design. 2014. What the World Eats. *National Geographic* (2014). <https://www.nationalgeographic.com/what-the-world-eats/>
- [17] Anna Flagg. 2017. The Opposite of Sanctuary. *The Marshall Project* (2017). <https://www.themarshallproject.org/2017/02/20/the-opposite-of-sanctuary>
- [18] Anna Flagg. 2018. The Myth of the Criminal Immigrant. *The Marshall Project* (2018). <https://www.themarshallproject.org/2018/03/30/the-myth-of-the-criminal-immigrant?ref=hp-1-112>
- [19] Walter Frick. 2016. The Decline of Yahoo in Its Own Words. *Harvard Business Review* (2016). <https://hbr.org/2016/06/the-decline-of-yahoo-in-its-own-words>

- [20] Russell Goldenberg. 2018. The World Through the Eyes of the US. *The Pudding* (2018). <https://pudding.cool/2018/12/countries/>
- [21] Russell Goldenberg and Amber Thomas. 2019. How Many High School Stars Make It in the NBA? *The Pudding* (2019). <https://pudding.cool/2019/03/hype/>
- [22] Kirk Goldsberry. 2019. How Mapping Shots In The NBA Changed It Forever. *FiveThirtyEight* (2019). <https://53eig.ht/2PF20gE>
- [23] Jackie Gu. 2018. Women Lose Out to Men Even Before They Graduate From College. *Bloomberg* (2018). <https://www.bloomberg.com/graphics/2018-women-professional-inequality-college/>
- [24] Eelke Heemskerk. APRIL 21, 2016. How Corporate Boards Connect, in Charts. *Harvard Business Review* (APRIL 21, 2016). <https://hbr.org/2016/04/how-corporate-boards-connect-in-charts>
- [25] Walt Hickey and Gus Wezerek. 2016. The Definitive Analysis Of ‘Love Actually,’ The Greatest Christmas Movie Of Our Time. *FiveThirtyEight* (2016). <http://53eig.ht/2he9BVh>
- [26] Josh Holder. 2018. How the NHS Winter Beds Crisis is Hitting Patient Care. *The Guardian* (2018). <https://www.theguardian.com/society/ng-interactive/2018/jan/11/how-the-nhs-winter-beds-crisis-is-hitting-patient-care>
- [27] Josh Holder and Alex Hern. 2018. Bezos’s Empire: How Amazon Became the World’s Most Valuable Retailer. *The Guardian* (2018). <https://www.theguardian.com/technology/ng-interactive/2018/apr/24/bezoss-empire-how-amazon-became-the-worlds-biggest-retailer>
- [28] Josh Katz, Kevin Quealy, and Margot Sanger-Katz. 2019. Would ‘Medicare for All’ Save Billions or Cost Billions? *The New York Times* (2019). <https://nyti.ms/2UFGVaV>
- [29] Ella Koeze and Neil Paine. 2019. The Story of the NBA Regular Season in 9 Charts. *FiveThirtyEight* (2019). <https://53eig.ht/2KHkkqL>
- [30] Niko Kommenda, Caelainn Barr, and Josh Holder. 2018. Gender Pay Gap: What We Learned and How To Fix It. *The Guardian* (2018). <https://www.theguardian.com/news/ng-interactive/2018/apr/05/women-are-paid-less-than-men-heres-how-to-fix-it>

- [31] Daniel Lathrop and Anna Flagg. 2017. Killings of Black Men by Whites are Far More Likely to be Ruled “Justifiable”. *The Marshall Project* (2017). <https://www.themarshallproject.org/2017/08/14/killings-of-black-men-by-whites-are-far-more-likely-to-be-ruled-justifiable>
- [32] Lauren Leatherby and Paul Murray. 2019. A Staggering Number of Candidates Are Running for U.S. President. *Bloomberg* (2019). <https://www.bloomberg.com/graphics/democratic-party-candidates-running-2020-election/>
- [33] Denise Lu and Karen Yourish. 2019. The Turnover at the Top of the Trump Administration. *The New York Times* (2019). <https://nyti.ms/2FQ0KBq>
- [34] Yolanda Martinez. 2018. Sending Even More Immigrants to Prison. *The Marshall Project* (2018). <https://www.themarshallproject.org/2018/05/20/sending-even-more-immigrants-to-prison>
- [35] Dave Merrill and Lauren Leatherby. 2018. Here’s How America Uses Its Land. *Bloomberg* (2018). <https://www.bloomberg.com/graphics/2018-us-land-use/>
- [36] Alicia Parlapiano and Jugal K. Patel. 2018. With Kennedy’s Retirement, the Supreme Court Loses Its Center. *The New York Times* (2018). <https://nyti.ms/2IyGAwh>
- [37] Adam Pearce and Dorothy Gambrell. 2016. This Chart Shows Who Marries CEOs, Doctors, Chefs and Janitors. *Bloomberg* (2016). <https://www.bloomberg.com/graphics/2016-who-marries-whom/>
- [38] Adam Pearce and Joe Ward. 2018. LeBron James Is Carrying the Cavaliers in a Historic Way. *The New York Times* (2018). <https://nyti.ms/2JojZ70>
- [39] Oliver Roeder. 2019. The Man Who Solved ‘Jeopardy!’. *FiveThirtyEight* (2019). <https://53eig.ht/2UzjXxS>
- [40] Simon Scarr and Marco Hernandez. 2019. A Network of Extremism Expands. *Reuters Graphics* (2019). <https://graphics.reuters.com/SRI%20LANKA-BLASTS-PLOTTER/010091W52YP/index.html>
- [41] NPR Staff. 2015. The Urban Neighborhood Wal-Mart: A Blessing Or A Curse? *NPR* (2015). <https://n.pr/1IP5XF2>
- [42] Ashlyn Still and Howard Schneider. 2018. Looking for Workers. *Reuters Graphics* (2018). <http://fingfx.thomsonreuters.com/gfx/rngs/USA-ECONOMY-JOBS/010062VB4V2/index.html>

- [43] Jessica Taylor, Katie Park, Tyler Fisher, and Alyson Hurt. 2017. Health Care Plan Championed By Trump Hurts Counties That Voted For Him. *NPR* (2017). <https://n.pr/2n1am50>
- [44] The Data Team. 2018. The Global Slump in Press Freedom. *The Economist* (2018). <https://www.economist.com/graphic-detail/2018/07/23/the-global-slump-in-press-freedom>
- [45] Amber Thomas. 2019. Sing My Name. *The Pudding* (2019). <https://pudding.cool/2019/05/names-in-songs/>
- [46] Amelia Thomson-DeVeaux and Gus Wezerek. 2019. Here's Why The Anti-Abortion Movement Is Escalating. *FiveThirtyEight* (2019). <https://53eig.ht/2WVfYh0>
- [47] Jason Treat and Anna Scalamogna. 2014. We'll Have What They're Having. *National Geographic* (2014). <https://www.nationalgeographic.com/foodfeatures/diet-similarity/>
- [48] Cory Turner, Jennifer Guerra, Sam Zeff, Kate Mcgee, Aaron Schrank, Jenny Brundin, Rob Manning, Ana Tintocalis, and Paul Boger. 2016a. Is There A Better Way To Pay For America's Schools? *NPR* (2016). <https://n.pr/1SPMXfA>
- [49] Cory Turner, Reema Khrais, Tim Lloyd, AlexANDra Olgin, Laura Isensee, Becky Vevea, and Dan Carsen. 2016b. Why America's Schools Have A Money Problem. *NPR* (2016). <https://n.pr/1p1NMag>
- [50] Nicole Washington, Jason Treat, John Kondis, and NG Staff. 2016. See Where Access to Clean Water Is Getting Better—and Worse. *National Geographic* (2016). <https://www.nationalgeographic.com/clean-water-access-around-the-world/>
- [51] Alex Webb and Chloe Whiteaker. 2016. Technology and Car Companies Are More Intertwined Than Ever. *Bloomberg* (2016). <https://www.bloomberg.com/graphics/2016-merging-tech-and-cars/>
- [52] Jin Wu, Weiyi Cai, and Simon Scarr. 2018. Oil Spilled at Sea: Putting the Sanchi Disaster Into Perspective. *Reuters Graphics* (2018). <https://graphics.reuters.com/OIL-SPILLS/010060SL1GQ/index.html>
- [53] Steven Yaccino, Jeremy Scott Diamond, and Mira Rojanasakul. 2015. This is Who Republican Presidential Contenders Follow on Twitter. *Bloomberg* (2015). <https://www.bloomberg.com/politics/graphics/2015-who-republican-candidates-follow/>

Appendix H

RESPONSIVE VISUALIZATION EXAMPLES

To better understand the responsive techniques used in news articles, we performed an open-coding process to analyze the techniques for 231 visualizations from 53 news articles. The results of this analysis are described in Section 8.3.1. In this section we provide some examples of the codes generated during the analysis process. We also show the reproductions produced using our responsive visualization system for seven visualization examples (Section 8.4).

H.1 “Total Cost of Major Natural Disasters”

This visualization is from the New York Times article “The Places in the U.S. Where Disaster Strikes Again and Again” [G13]. This visualization is a static, annotated bar chart that adapts the visualization size and annotations to different device contexts. In particular, this visualization exhibits a number of interesting changes between the desktop and mobile versions, primarily around the annotation strategy and title placement. To better understand the responsive techniques used, we performed an open-coding process of the changes between the desktop and mobile versions of the visualization. The codes we generated for this visualization are included in Figure H.8a. Using our responsive visualization system, we recreate and extend this visualization to include five device-specific designs (Figure H.1): a desktop visualization (1024 x 612px), a portrait phone visualization (375 x 325px), a landscape phone visualization (585 x 325px), a thumbnail (240 x 120px), and a square thumbnail (100 x 100px). A video walkthrough of this example is available on YouTube at the following link: <https://youtu.be/wIvh6UBfMW0>. This example is discussed in Section 8.4.1.

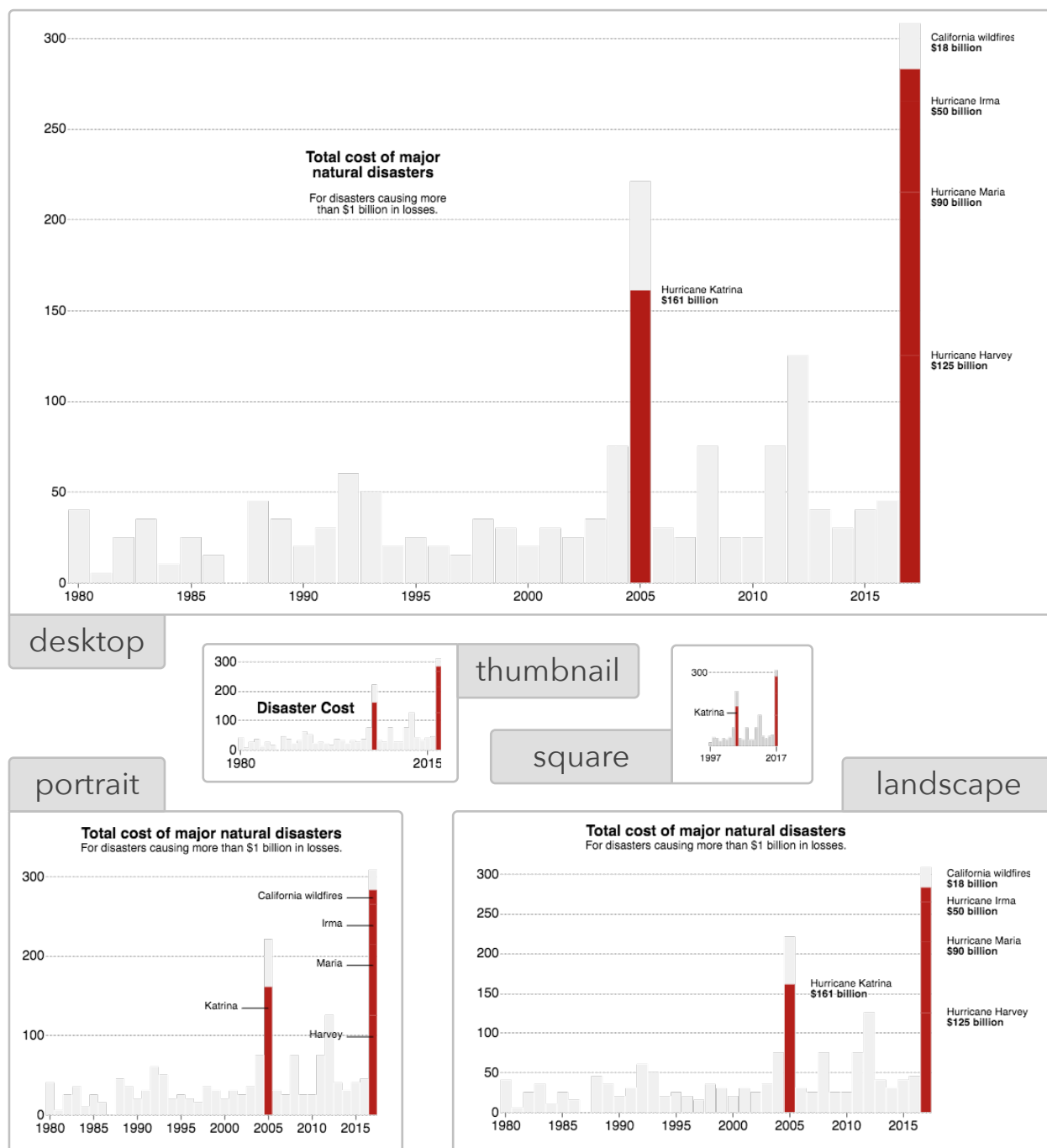


Figure H.1: Five visualizations created using our responsive visualization system based on the visualization “Total Cost of Major Natural Disasters” from the New York Times [G13]. The square thumbnail was submitted as part of the publication of this work at CHI 2020 [84].

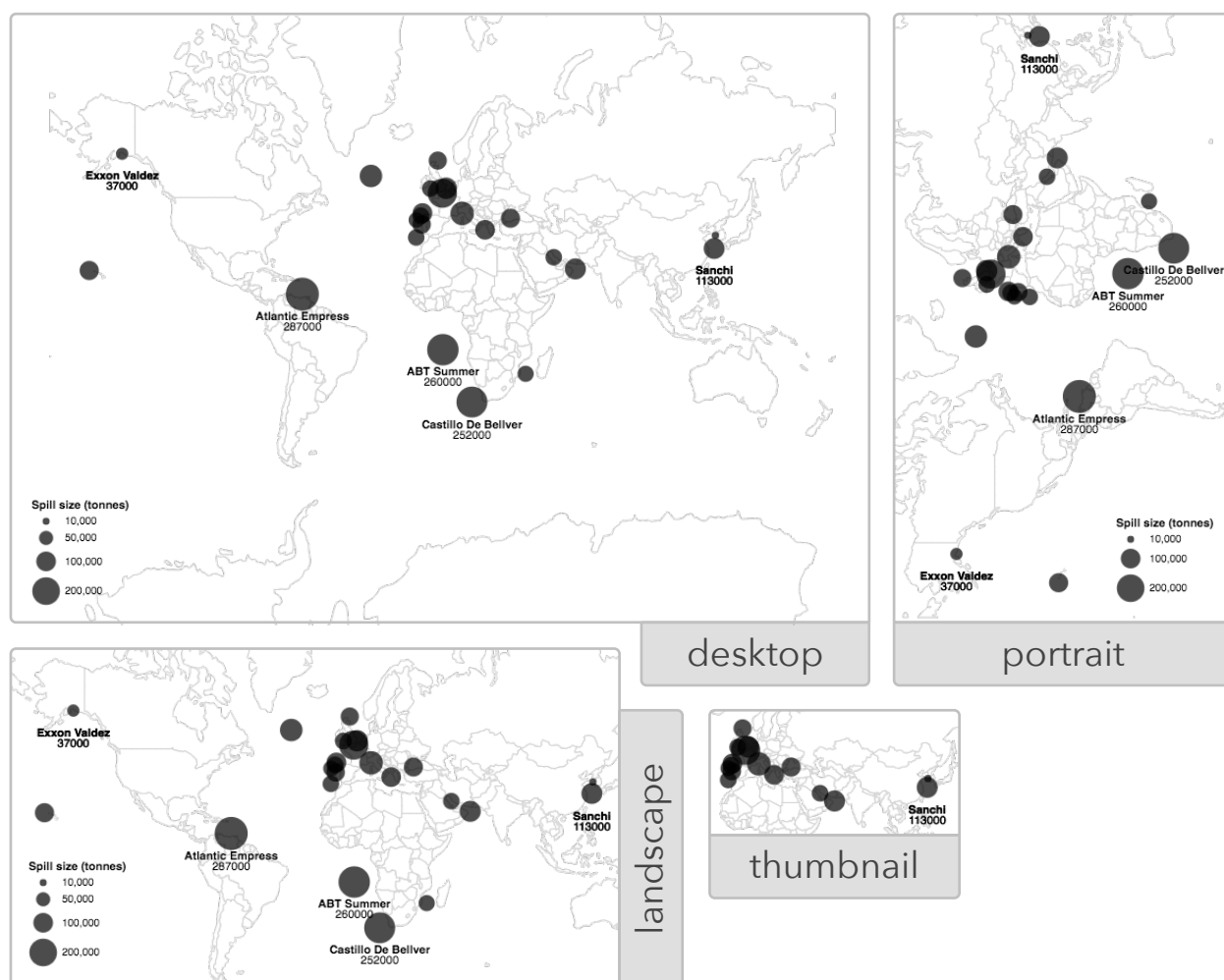


Figure H.2: Four visualizations created using our responsive visualization system based on the visualization “Incidents at Sea” from Reuters Graphics [G52].

H.2 “Incidents at Sea”

This visualization is from the Reuters article “Oil spilled at sea: Putting the Sanchi disaster into perspective” [G52]. This visualization is a static, annotated symbol map that adapts the visualization size and annotations to different device contexts. This article includes a map visualization that reduces the number of text annotations, rescales the text, and updates the size encoding and legend between different device-specific visualizations. To better understand the responsive techniques used, we performed an open-coding process of the changes between the desktop and mobile versions of the visualization. The codes we generated for

this visualization are included in Figure H.8b. Using our responsive visualization system, we produce a map with similar annotations to the one from Reuters Graphics [G52], but also demonstrate other responsive techniques (e.g., rotating the map for the portrait orientation). We produce four designs (Figure H.2): a desktop visualization (825 x 585px), a portrait phone visualization (325 x 585px), a landscape phone visualization (585 x 325px), and a thumbnail (240 x 120px). The desktop visualization approximates the one from Reuters Graphics [G52], using data from ITOPF [98], which includes the top 20 major spills through 2018; as such, some of the data points from the Reuters graphic are not included in this recreation. We rotate the map to better use the space of the portrait orientation. However, we maintain the rotation of the labels to ensure legibility of the text. For this example, the landscape version is very similar to the desktop version, but with a much tighter margin on the map content; however, the map still shows all the same data. Since the original article focuses on comparing to the Sanchi disaster, we designed the thumbnail to focus on this piece of data with a hint of the larger context. A video walkthrough of this example is available on YouTube at the following link: <https://youtu.be/GwnyMg9QnyM>. The walkthrough and rationale for this example is further discussed in Section 8.4.2.

H.3 “In close decisions, Kennedy voted in the majority...”

This visualization is from the New York Times article “With Kennedy’s Retirement, the Supreme Court Loses Its Center” [G36]. This visualization includes two drastically different designs (a horizontal dot plot and a vertical bar chart) depending on the device context. Both visualizations are static. To better understand the responsive techniques used, we performed an open-coding process of the changes between the desktop and mobile versions of the visualization. The codes we generated for this visualization are included in Figure H.8c. Using our responsive visualization system, we recreate and extend this visualization to include three device-specific designs (Figure H.3): a desktop visualization (810 x 418px), a portrait phone visualization (325 x 585px), and a thumbnail (240 x 120px). For the desktop visualization, the New York Times uses a dot plot visualization to show the percent of times each justice

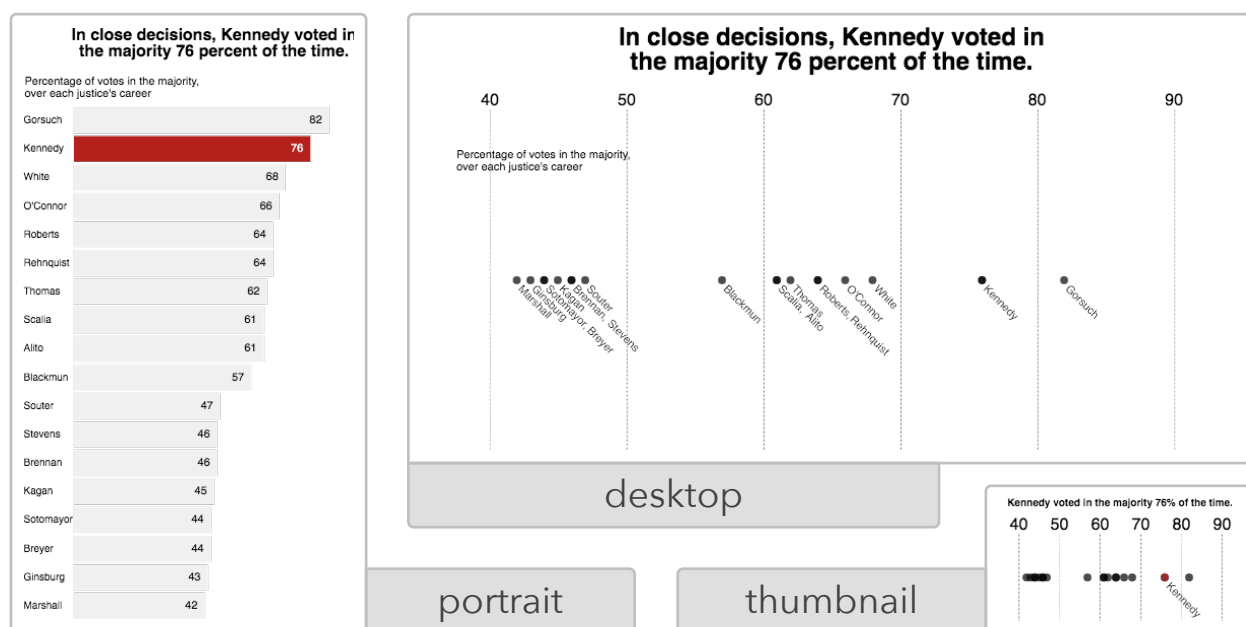


Figure H.3: Three designs created using our responsive visualization system from a New York Times visualization: “In close decisions, Kennedy voted in the majority 76 percent of the time” [G36].

votes in the majority. For the desktop visualization, the main difference in our recreation is that we are missing the yellow background to highlight the text for Kennedy. The landscape visualization exhibits a drastic change on mobile, in which the dot plot changes to a bar chart; the underlying data for each visualization is the same. For the thumbnail, we use the desktop visualization as a base, and update the title for the smaller space; we also reduce the number of labels. A video walkthrough of this example is available on YouTube at the following link: <https://youtu.be/zQn1URMLSJU>. This example is discussed in Section 8.4.3.

H.4 “Percentage of the population without access to improved water”

This visualization is from the National Geographic article “See Where Access to Clean Water Is Getting Better – and Worse” [G50]. This visualization is an interactive combination line chart and scatterplot showing the different countries and regions of interest; in the mobile version, only the regions are shown. Both versions of the visualization support interaction via a dropdown. To better understand the responsive techniques used, we performed an open-coding process of the changes between the desktop and mobile versions of

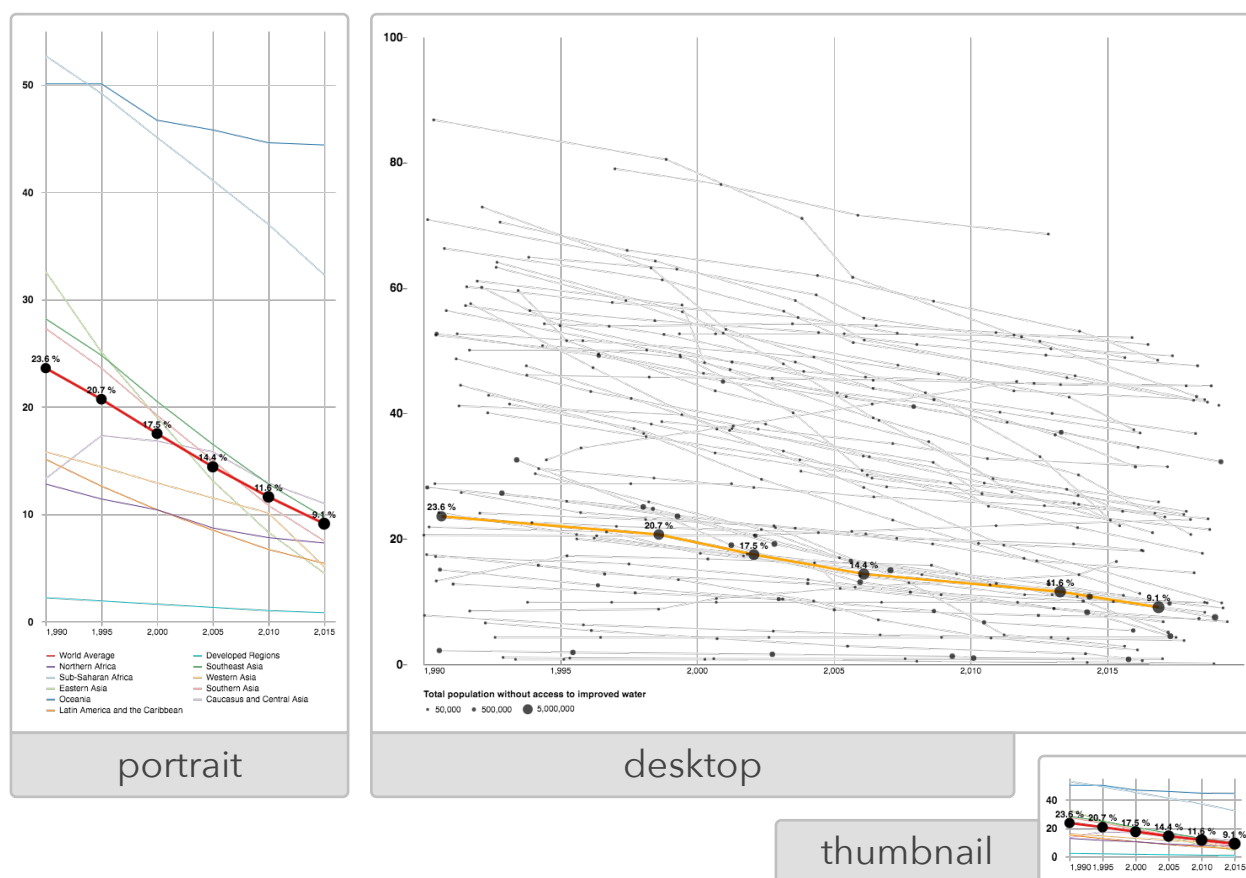


Figure H.4: Three designs created using our responsive visualization system based on a National Geographic visualization: “Percentage of the population without access to improved water” [G50].

the visualization. The codes we generated for this visualization are included in Figure H.8d. Using our responsive visualization system, we recreate and extend this visualization to include three device-specific designs (Figure H.4): a desktop visualization (1024 x 812px), a portrait phone visualization (375 x 812px), and a thumbnail (240 x 120px). For our recreations, we reduce the number of data points to limit the impact on the performance of our system and the visualization rendering. This example is particularly exciting because it is interactive; for the screenshots shown here, we select the “World Average” from the dropdown. A video walkthrough of this example is available on YouTube at the following link: <https://youtu.be/2qs3--pYn-o>. This example is further discussed in Section 8.4.4.

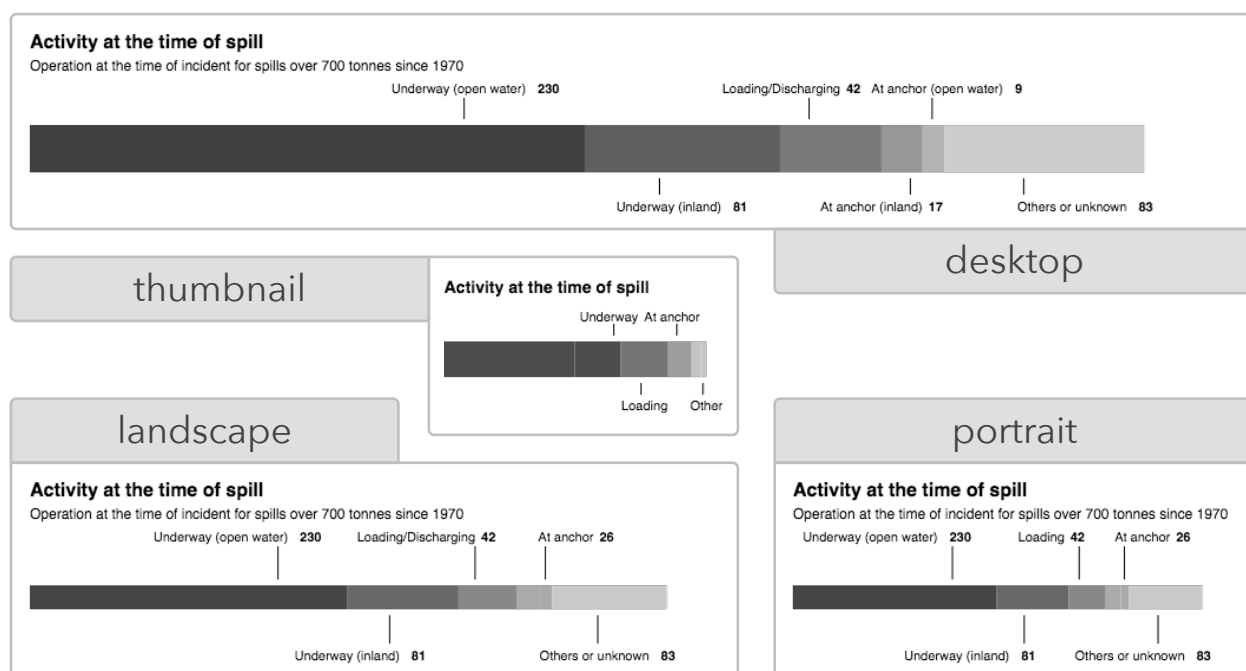


Figure H.5: Four designs created using our responsive visualization system based on a Reuters Graphics visualization: “Activity at the time of spill” [G52].

H.5 “Activity at the time of spill”

This visualization is from the Reuters Graphics article “Oil spilled at sea: Putting the Sanchi disaster into perspective” [G52]. This visualization is a static, annotated stacked bar chart that adapts the visualization size and annotations to different device contexts. To better understand the responsive techniques used, we performed an open-coding process of the changes between the desktop and mobile versions of the visualization. The codes we generated for this visualization are included in Figure H.8e. This example is particularly interesting because the different versions change the binning scheme for the visualized data to show between four and six categories. Using our responsive visualization system, we recreate and extend this visualization to include four device-specific designs (Figure H.5): a desktop visualization (1024 x 150px), a landscape phone visualization (585 x 150px), a portrait phone visualization (375 x 150px), and a thumbnail (240 x 120px).

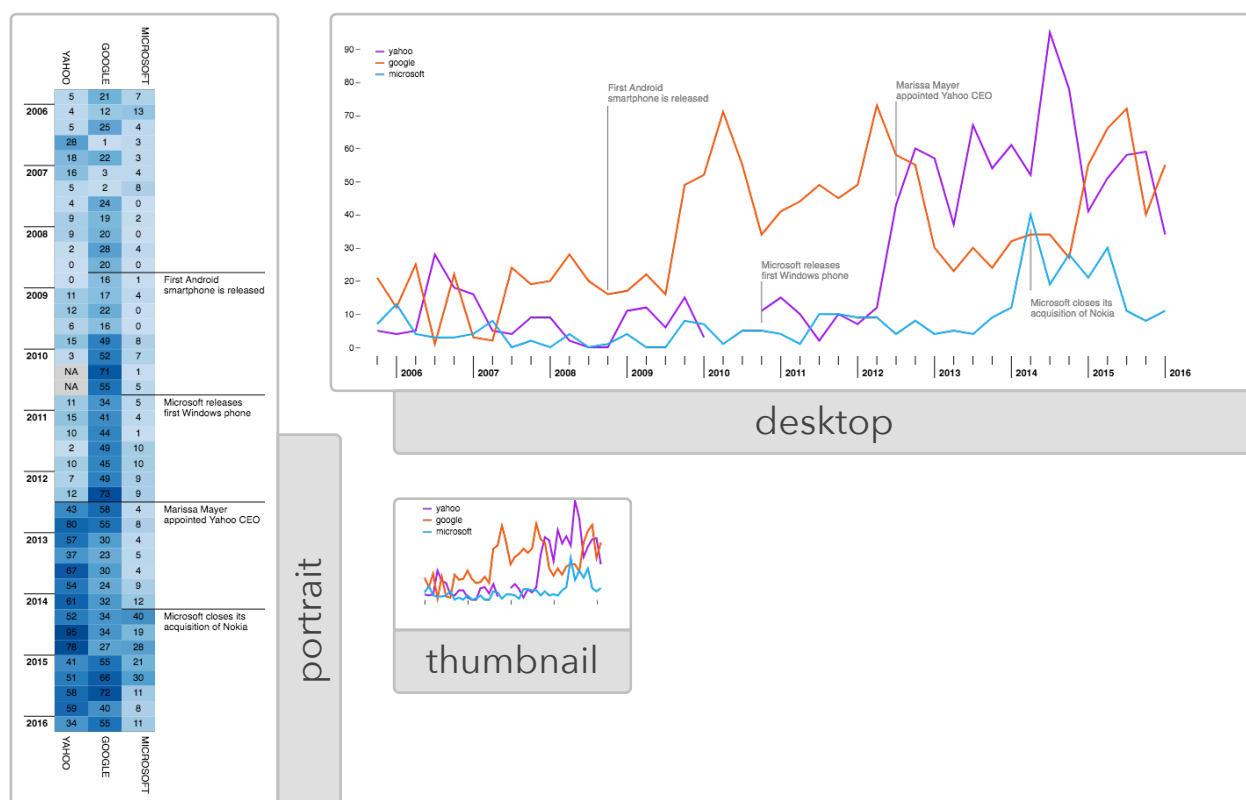


Figure H.6: Three designs created using our responsive visualization system based on a Harvard Business Review visualization: “Was Yahoo Late to Mobile?” [G19].

H.6 “Was Yahoo Late to Mobile?”

This visualization is from the Harvard Business Review article “The Decline of Yahoo in Its Own Words” [G19]. This visualization includes two drastically different designs (a line chart and a heatmap) depending on the device context. Both visualizations are static and both are annotated with the same information. The codes generated during the open-coding process are included in Figure H.8g. Using our responsive visualization system, we recreate and extend this visualization to include three device-specific designs (Figure H.6): a desktop visualization (1024 x 400px), a portrait phone visualization (275 x 875px), and a thumbnail (240 x 120px). The desktop visualization uses a multi-series line chart with annotations that fits well on standard screen sizes. The phone visualization uses a heatmap to better fit the portrait orientation, but also requires users to scroll to see all of the visualization content.

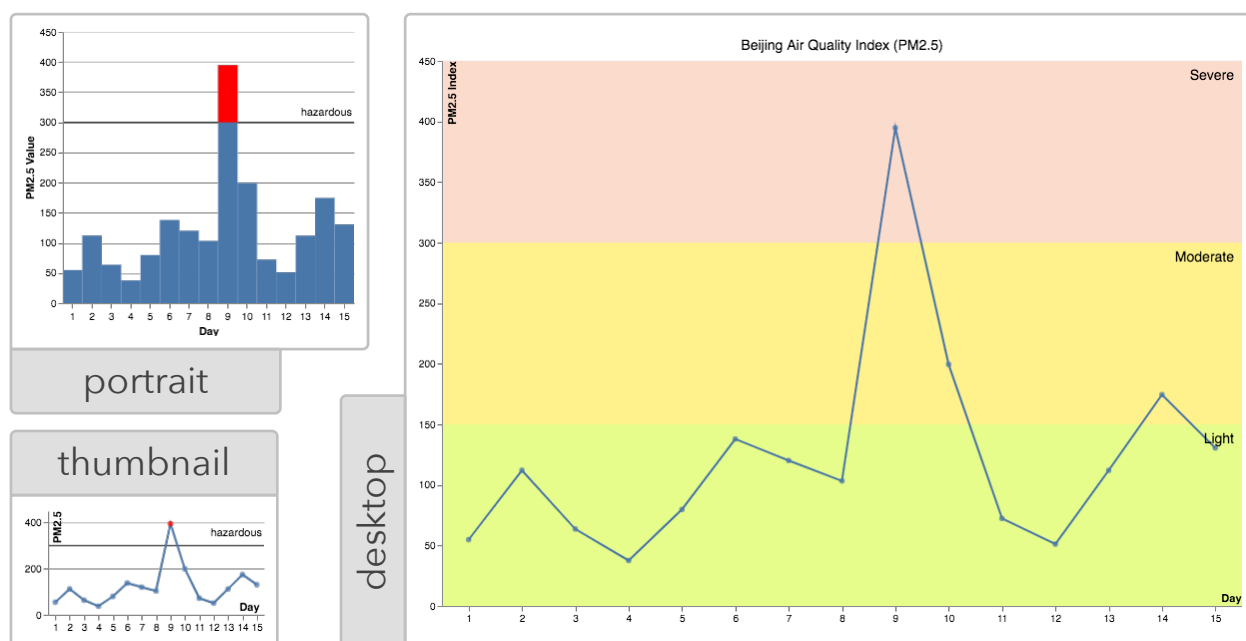


Figure H.7: Three designs created using our responsive visualization system based on an example from ChartAccent [159] and Vega-Lite [176].

H.7 “Beijing Air Quality Index (PM2.5)”

This visualization is based on data from ChartAccent [159] showing the Beijing Air Quality Index (PM2.5) of December 2014. The provided sample data was used for a Vega-Lite [176] example to show a simple layered bar chart visualization. Note that this example was *not* included in the original responsive visualization corpus. Therefore, the codes shown here (Figure H.8f) only correspond to our reproduced example and are not included as part of the counts in the original paper, as described in Section 8.3.1. Using our responsive visualization system, we recreate and extend this visualization to include three designs (Figure H.7): a desktop visualization (825 x 585px), a portrait phone visualization (332 x 310px), and a thumbnail (240 x 120px). The desktop version is based on a sample visualization from ChartAccent [159], which shows an annotated line chart of PM2.5 values in December 2014; this recreation uses the smaller dataset from the Vega-Lite visualization. The portrait phone visualization is a recreation of one of the Vega-Lite [176] examples: an bar chart of PM2.5 values, with annotated hazardous air levels. The thumbnail combines features of the other two visualizations to show a line chart and annotate the threshold for hazardous values.

(a) Total Cost of Major Natural Disasters		
Desktop vs. Mobile (Portrait)		
Code	Action	Component
reposition title text	reposition	title
shorten annotations	modify	labels
remove annotation details	modify	labels
shorten axis year labels	modify	axis labels
reduce bar width	resize	marks
width compressed	resize	view
align axis labels	reposition	axis labels
remove wrap title text	reposition	title
y axis gridline changes	resize	gridlines
remove y axis ticks	remove	axis ticks
Desktop vs. Mobile (Landscape)		
Code	Action	Component
no changes	no changes	view

(b) Incidents at Sea		
Desktop vs. Mobile (Portrait)		
Code	Action	Component
scale visualization smaller	resize	view
remove some text annotations	remove	labels
change size encoding	resize	marks
reduce legend marks	modify	legend
Desktop vs. Mobile (Landscape)		
Code	Action	Component
scale visualization smaller	resize	view

(c) In close decisions, Kennedy voted in the majority...		
Desktop vs. Mobile (Portrait)		
Code	Action	Component
change mark encoding	modify	marks
label mark percentage	add	labels
remove percentage axis	remove	axis
swap encoding axes	reposition	marks
Desktop vs. Mobile (Landscape)		
Code	Action	Component
no changes	no changes	view

(d) Percentage of the population without access to...		
Desktop vs. Mobile (Portrait)		
Code	Action	Component
wrap title text	reposition	title
remove circle marks	remove	marks
reduce line marks	remove	marks
remove circle size legend	remove	legend
add color legend	add	legend
change grid type	reposition	view
compress width	resize	view
remove some axis labels	remove	axis labels
add gridlines	add	gridlines
remove animation	remove	interaction
remove rural/urban widget	remove	interaction
reduce dropdown options	modify	interaction
Desktop vs. Mobile (Landscape)		
Code	Action	Component
disabled	remove	view
disabled	remove	interaction

(e) Activity at the time of spill		
Desktop vs. Mobile (Portrait)		
Code	Action	Component
compress width	resize	view
reposition text annotations	reposition	labels
simplify binning	modify	data
Desktop vs. Mobile (Landscape)		
Code	Action	Component
compress width	resize	view
overlap annotation text	reposition	labels

(f) Was Yahoo Late to Mobile?		
Desktop vs. Mobile (Portrait)		
Code	Action	Component
change visualization design	modify	view
wrap subtitle text	reposition	title
modify color encoding	modify	marks
swap axes	reposition	marks
reposition text annotations	reposition	labels
remove axis ticks	remove	axis ticks
remove count axis	remove	axis
add count labels	add	labels
add company axis	add	axis
Desktop vs. Mobile (Landscape)		
Code	Action	Component
change visualization design	modify	view
wrap subtitle text	reposition	title
modify color encoding	modify	marks
swap axes	reposition	marks
reposition text annotations	reposition	labels
remove axis ticks	remove	axis ticks
remove count axis	remove	axis
add count labels	add	labels
add company axis	add	axis

(g) Beijing Air Quality Index (PM2.5)		
Desktop vs. Mobile (Portrait)		
Code	Action	Component
compress width	resize	view
compress height	resize	view
change mark type	modify	marks
remove region highlights	remove	marks
remove region labels	remove	labels
remove title	remove	title
reposition axis title	reposition	axis title
add gridlines	add	gridlines
add bar highlight	add	marks
add threshold mark	add	marks
add threshold annotation	add	labels

Figure H.8: The responsive visualization codes for each visualization described in this section, for both the portrait and landscape orientation of the phone. (a) “Total Cost of Major Natural Disasters” from the New York Times [G13]. (b) “Incidents at Sea” from Reuters Graphics [G52]. (c) “In close decisions, Kennedy voted in the majority 76 percent of the time” from the New York Times [G36]. (d) “Percentage of the population without access to improved water” from National Geographic [G50]. (e) “Activity at the time of spill” from Reuters Graphics [G52]. (f) “Was Yahoo Late to Mobile?” from the Harvard Business Review [G19]. (g) “Beijing Air Quality Index (PM2.5),” based on examples from ChartAccent [159] and Vega-Lite [176].