

My research explores the design of new interactive systems and programming languages that help people better manage, author, understand, and reuse both code and data. 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.

VISUALIZING & UNDERSTANDING THE BEHAVIOR OF REACTIVE PROGRAMMING LANGUAGES

Consider the design of Vega [1]: a declarative grammar for producing interactive visualizations. In Vega, programmers author a new visualization by focusing on important encoding decisions—what data should be visualized and in what way—rather than the low-level implementation details (e.g., how the visualization is programmatically assembled). Programmers may similarly specify the interactive capabilities of the visualization (e.g., how potential viewers can explore the results). However, this approach introduces a gap between the code the programmer writes and the system output: the user is often required to maintain a complex mental model of the underlying program behavior to effectively understand and debug it. This separation becomes particularly problematic when trying to fully understand the interactive behaviors of the resulting visualization.

Problem: Programmers struggle to overcome the gap between high-level code and system output.


Solution: New visualization tools [2,3] automatically surface relevant details of the program behavior in real-time within the context of the code, enabling users to focus on their primary development task.

In early formative interviews with Vega programmers, I found that programmers primarily require debugging tools that mirror the level of abstraction with which they are already familiar. Rather than trying to educate users about underlying details of the system architecture, new tools should focus on presenting actionable information in an appropriate context. Following these guidelines, I developed several novel approaches to program visualization [2,3] using Vega as a petri dish.

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

Figure 1: A Vega code snippet augmented with inline visualizations showing the distribution of data for different array variables. For example, the “symbol” variable is an array containing five unique string representing different companies, one of which occurs less frequently than others.

These visualizations provide a holistic view of the program behavior by **automatically visualizing the state and history of all program variables**. These program visualizations update in real-time as the programmer interacts with the output Vega visualization and are presented alongside the context of the code. My first approach introduced (1) a timeline of interaction events and variable updates that enables users to record and replay interactions with the output, (2) dynamic data tables showing the distribution and variation of data for the underlying datasets, and (3) a tooltip to probe data transformations and encodings directly on the output visualization [2]. The timeline and a simplified data view are currently employed in the online Vega editor [4]. To identify the impact of this approach, I conducted a lab study with 12 novice

Vega programmers who were unfamiliar with both the specific code and the Vega programming language; I found that participants could **effectively understand the source code to identify bugs or crucial dependencies**, but found that the environment still required expertise to effectively navigate. In my follow-up work [3], program visualizations are displayed directly inline  within source code (Figure 1), allowing programmers to inspect the history of values for program variables. This approach unloads the burden of recalling or mentally tracking the program state from the user to the inline visualizations. Instead of switching between multiple views to author, test, and debug their code, programmers can **focus on editing and debugging code while viewing the behavior of program variables inline**. In an evaluation with 18 novice Vega programmers, participants had more correct answers to program understanding questions when using the inline visualizations compared to a baseline condition; the inline visualizations also improved participants' self-reported speed and accuracy.

FUTURE WORK ON PROGRAM (AND SYSTEM) VISUALIZATION:

Vega’s reactive semantics enabled me to efficiently snapshot the program state at every point in the execution history; furthermore, I was able to utilize the structure of the Vega code to easily identify and visualize all program variables. While this work exemplifies how the techniques may be applied for a large class of reactive programming languages, I would like to further explore how to effectively incorporate real-time program visualizations into imperative programming domains. In order to appropriately incorporate inline visualizations for real-world use cases, I would like to further explore how the style and placement of inline visualizations could dynamically adapt based on the user’s current task. Finally, these visualizations aim to enhance the programming experience, but could be useful for augmenting end-user systems with relevant or timely information. For example, inline visualizations could be employed in visualization construction systems like Tableau [5], Data Illustrator [6], or Lyra [7] to surface information about the underlying data or data transformation pipelines, and thus better support how users understand and manipulate data to produce their desired visualization design.

LIGHTWEIGHT DESIGN OF CUSTOMIZED, DOMAIN-SPECIFIC GRAPH LAYOUTS WITH SETCOLA

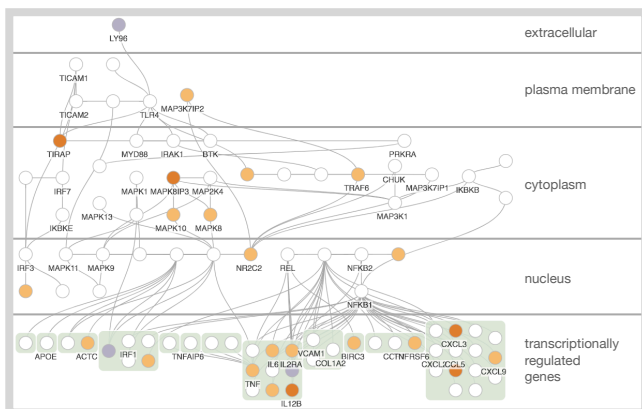


Figure 2: The layout for the TLR4 biological system produced using only eight constraints in SetCoLa. Layers in the graph correspond to the location of biomolecules in a cell.

To better explore or understand their data, domain experts often create complex graph visualizations by leveraging domain-specific properties of the data to inform the visualization layout. For example, customized layouts are commonly used to visualize biological systems based on knowledge of the cellular structure (Figure 2), or to track and understand the spread of disease based on demographics, location, or other relevant properties. However, common approaches to domain-specific graph layout often require domain experts to either use ill-fitting techniques that do not appropriately reflect their needs, or to develop new tools specifically designed for their particular use case.

Problem: Domain experts lack the tools necessary to produce custom, domain-specific graph layouts.

Solution: SetCoLa [8] enables high-level constraint based layouts that require an order of magnitude fewer user-authored constraints than previous approaches and support layout reuse across similar graphs.

To better support domain experts in designing custom graph layouts, I developed SetCoLa [8]: a new programming language for specifying high-level constraints for graph layout based on existing domain-specific properties of the graph (e.g., node attributes and topology). Whereas prior approaches utilized node-level constraints between individual pairs of nodes, SetCoLa **reduces the number of constraints written by the user by one to two orders of magnitude**. The layout in Figure 2 requires only eight constraints in SetCoLa compared to the 363 constraints generated for the underlying constraint engine, WebCoLa [9]. To evaluate the expressiveness of SetCoLa for customized layout, I reproduced three real-world examples and compared the size of the SetCoLa specification to the number of constraints required by the underlying constraint solver. SetCoLa can **facilitate program understanding by representing constraints in terms of domain-specific properties familiar to the user**. Since constraints are defined based on domain-specific properties of the graph, SetCoLa additionally enables **reuse of custom layouts** across graphs within the same domain. I demonstrated this functionality with an application of a custom biological system layout across multiple graphs extracted from InnateDB [10].

FUTURE WORK ON CONSTRAINTS, GRAPH LAYOUT, AND UNDERSTANDING SYSTEMS:

Constraints are a flexible and powerful approach to solving complex problems such as custom layout. However, the execution and invalidation of constraints can be hard to comprehend in such complex systems, and techniques to support the process of understanding or debugging constraint systems is currently limited. In the future, I would like to explore the design of new techniques to facilitate program understanding for constraints. In SetCoLa, by connecting constraints directly to domain-specific properties of the graph, we can better reflect the intentions and expertise of the user. I would like to further explore how best to communicate the user’s intent in the system and translate the system output back to actionable information for the user. This future work could prove integral to the application and widespread use of constraints. Going further, I want to explore how communicating both user and system intent can positively impact a larger class of complex end users systems.

AUTHORING AND REUSING RESPONSIVE VISUALIZATION DESIGNS FOR MOBILE DEVICES

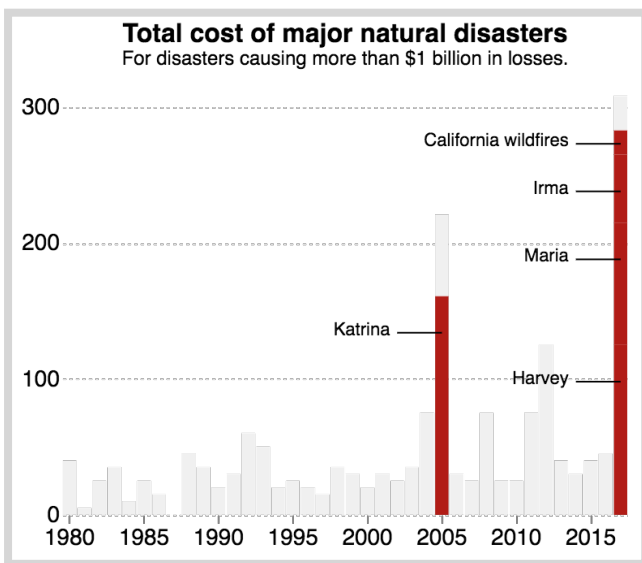


Figure 3: A bar chart visualization for small mobile devices reproduced from the New York Times Article “The Places in the U.S. Where Disaster Strikes Again and Again” [12].

Readers more often consume media content on a phone than a desktop computer. Therefore, news outlets must be able to appropriately adapt content based on the device that is used. While text content can adapt to the device size via reflow, it is non-trivial to create responsive visualizations. Responsive visualizations must adapt their design so that content remains informative and legible across different device contexts and visualization sizes. For example, visualizations might be displayed as a figure (Figure 3) or inline as a sparkline depending on the amount of space available. To produce responsive visualizations, designers may choose to resize certain visualization marks, swap the axis encodings so that a chart fits better on a mobile screen, or remove unessential labels. However, the process of effectively designing and maintaining device-specific visualizations can prove time-consuming and labor intensive.

Problem: Journalists utilize an inflexible design workflow for creating responsive visualizations.

Solution: A new system for responsive visualization design [11] demonstrates a core set of system features that enable simultaneous editing, device-specific customization, and flexible workflows for responsive visualization design by foregrounding variation across designs to summarize differences.

In formative interviews with five journalists, I found that responsive design was often forefront in their minds, yet most designers focused on desktop development first and customized designs for mobile as the last step in the development process. This decision was motivated primarily by systems that were geared towards desktops first, rather than reflecting the designers' development preferences. Furthermore, I found that designers often discarded ideas based on presumptions about how the design would work on mobile without ever exploring the reality of the design.

To enable more flexible design workflows with an emphasis on mobile (in contrast to the linear, desktop-first process described by journalists), I developed a new visualization construction system that allows designers to simultaneously view, create, and modify multiple device-dependent visualizations via linked editing [11]. This system displays separate views for each chart size of interest and **foregrounds the variation between visualizations to help designers assess the full picture of the customizations applied to individual views**. Users may also propagate changes across views by synching the designs to remove undesirable variations. To evaluate this work, I describe the iterative processes for recreating four real-world responsive visualizations found in online news articles. These processes demonstrate the benefits of displaying multiple design versions and allowing designers to **freely move between editing different designs**.

FUTURE WORK ON RESPONSIVE VISUALIZATION DESIGN:


Responsive designs adapt the content to the specific device context being used. Another form of adaptation would be to adapt to the individual viewer observing the content. However, this approach would further complicate the design process for journalists keen to ensure that each design meets the standards of their organization. In future work, I would like to explore options for automatically adapting the visualization content based on device or user context. One option could be the use of constraints for reflecting the expectations of the user, and raises issues of program understanding and debugging as explored in my prior work. Since ensuring customizability and control are essential, I would also like to explore new techniques for summarizing multiple designs and supporting designers in exploring the space of visualizations that they create.

REFLECTION & FUTURE RESEARCH AGENDA

My research aims to better understand people and to help people better understand systems. For each project, I sought to understand the pain points faced by individuals in their current development process, and then identify and develop ways in which to mitigate the challenges that arose. I further evaluated the proposed approaches via user studies or by reproducing real-world examples to demonstrate the benefits provided by the proposed techniques, as well as new areas for future work.

Across each project, the approaches employed reflect the expertise and expectations of the end user. To help end user programmers of Vega, I visualize relevant system details at the level of abstraction they are familiar with [2,3]. Such techniques could further be applied to SetCoLa [8] to better highlight connections between individual node properties, the user-authored constraints, and/or the system produced constraints and layout. These techniques could also apply to end user visualization construction systems like the one I developed for responsive visualization [11]; in doing so, the system

could better surface details of the automatic layout decisions to support users in developing more dynamic responsive visualization designs. To produce such automated responsive designs may require the application of constraints. As in SetCoLa [8], these constraints should aim to reflect properties of interest to the user and should be generalizable across common visualization instances or customizations.

My projects to date focus on individual components of a development process and often include several assumptions about how the user is interacting with the system. One common assumption across every project is that the data will arrive in a clean, ready-to-use format. While minor transformations are supported in my responsive visualization system [11], this work primarily expects that the data will be ready to go. Similarly, SetCoLa [8] requires a specific graph format pre-processed to include all domain-specific properties of interest, with only minor adaptations available on demand. For Vega [1], a variety of data transformations are possible and my programming understanding systems aim to give insight into how those transformations behave or what happens when they fail [2,3]. For example, users can probe points in the output to see how data maps to the visual encodings (e.g., the color) `versicolor = c => #ff7f0e` [2] or to see the variation in a dataset  based on the behavior of data transformations over time [3]. These approaches surface details of the otherwise opaque data processing pipeline in Vega. In my future work, I would like to explore how these techniques or new approaches can better support users in developing and integrating data transformations into their design process for different end user systems and data analysis workflows.

When faced with a new project, users often interact with numerous different elements along the development pipeline, including but not limited to (1) creating, identifying, and cleaning data, (2) selecting the right system or approach to analyze the work, (3) understanding the system output and debugging the behavior in the face of erroneous or unexpected results, and (4) communicating important insights to themselves or others. Each component of this process may require a different set of expertise and a careful understanding of how each piece impacts the downstream results. In my future work, I would like to 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, I seek to design novel methods and tools that offset the burden on users while adapting to their changing needs and available resources.

REFERENCES

- [1] Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *Proc. InfoVis 2016*. Arvind Satyanarayan, Ryan Russell, **Jane Hoffswell**, Jeffrey Heer. <https://doi.org/10.1109/TVCG.2015.2467091>
- [2] Visual Debugging Techniques for Reactive Data Visualization. *Proc. EuroVis 2016*. **Jane Hoffswell**, Arvind Satyanarayan, Jeffrey Heer. <https://doi.org/10.1111/cgf.12903>
- [3] Augmenting Code with In Situ Visualizations to Aid Program Understanding. *Proc. CHI 2018*. **Jane Hoffswell**, Arvind Satyanarayan, Jeffrey Heer. <https://doi.org/10.1145/3173574.3174106>
- [4] Online Vega Editor. <https://vega.github.io/editor/>
- [5] Tableau. <https://www.tableau.com/>
- [6] Data Illustrator: Augmenting Vector Design Tools with Lazy Data Binding for Expressive Visualization Authoring. *Proc. CHI 2018*. Zhicheng Liu, John Thompson, Alan Wilson, Mira Dontcheva, James Delorey, Sam Grigg, Bernard Kerr, John Stasko. <https://doi.org/10.1145/3173574.3173697>
- [7] Lyra: An Interactive Visualization Design Environment. *Proc. EuroVis 2014*. Arvind Satyanarayan, Jeffrey Heer. <https://doi.org/10.1111/cgf.12391>
- [8] SetCoLa: High-Level Constraints for Graph Layout. *Proc. EuroVis 2018*. **Jane Hoffswell**, Alan Borning, Jeffrey Heer. <https://doi.org/10.1111/cgf.13440>
- [9] WebCoLa. <https://ialab.it.monash.edu/webcola/>
- [10] InnateDB. <https://www.innatedb.com/>
- [11] Techniques for Flexible Responsive Visualization Design. *Proc. CHI 2020*. **Best Paper Award (Top 1%)**. **Jane Hoffswell**, Wilmot Li, and Zhicheng Liu. To appear: <https://doi.org/10.1145/3313831.3376777>
- [12] The Places in the U.S. Where Disaster Strikes Again and Again. Sahil Chinoy. New York Times. <https://nyti.ms/38VBlzz>