EuroRV3: EuroVis Workshop on Reproducibility, Verification, and Validation in Visualization (2015)
W. Aigner, P. Rosenthal, and C. Scheidegger (Editors)

Debugging Vega through Inspection of the Data Flow Graph

Jane Hoffswell,[1] Arvind Satyanarayan[2] and Jeffrey Heer[1]

[1]University of Washington, [2]Stanford University

Abstract

Vega is a declarative visualization grammar that decouples specification from execution to allow users to focus on the visual representation rather than low-level implementation decisions. However, this representation comes at the cost of effective debugging as its execution is obfuscated. By presenting the developer with Vega's data flow graph along with interactive capabilities, we can bridge the gap between specification and execution to enable direct inspection of the connections between each component. This inspection can augment the developer's mental model of the specification, enabling the developer to more easily identify areas of interest and implement changes to the resulting visualization.

Categories and Subject Descriptors (according to ACM CCS): D.2.2 [Software Engineering]: Design Tools and Techniques—User interfaces

1. Introduction

Declarative languages can accelerate the development process by decoupling specification from execution [HB10]. This separation enables rapid iteration and exploration as developers only need to update the specification rather than reformulating the control flow. Similarly, declarative languages promote code reuse as the specification can easily be retargeted to a new dataset or augmented with different interaction techniques. By separating the execution, the language developers can implement new optimizations without inhibiting the designer's process.

However, declarative languages have a trade-off between flexibility and comprehensibility. Separating specification from execution obfuscates the underlying program state and inhibits the developer's ability to evaluate and debug the output. D3 treads this line by enabling declarative specification of visualization properties within a development environment which supports native debugging strategies [BOH11]. D3 leverages existing browser-based developer tools for debugging and uses an immediate evaluation strategy to reduce the mismatch between internal control flow and the developer's mental model. However, this direct inspection requires domain expertise and a clear mental model of the program execution.

Vega is a visualization grammar that builds on D3 with a higher-level JSON-based specification language [Veg14]. The simplified specification enables iterative exploration of the design space but limits access to the code execution. Identifying how changes to the specification impact the visualization, or how user interactions with the visualization and data are propagated through the execution, is therefore difficult. Visual representations of program state can provide developers with the context necessary to better interpret and interact with their code, but standard visualization techniques often lack scalability or limit the range of questions a developer could answer [Guo13, LBM14]. This paper presents a vision of how developers using Vega could visually inspect the underlying execution process. This inspection can help alleviate misconceptions about the translation between specification and visualization, and demonstrate how changes are realized in the resulting visualization.

2. Visualizing the Vega Data Flow Graph

Vega provides a high-level declarative grammar for producing common visualizations, and has been extended to support declarative interaction design [SWH14]. Vega accepts a JSON specification that is parsed into a data flow graph representing the execution pipeline. Data tuples are pushed through this data flow graph to be rendered into the final visualization. While these internals can be partially inspected via the JavaScript console, such inspection requires knowledge of the underlying structure of the Vega execution cycle. For developers without this expertise, the specification and resulting visualization are the primary resources for debugging, but provide no insight into this structure (Figure 1a, b). A visualization of the underlying Vega execution model can bridge this gulf.

Consider a scenario in which a developer wants to implement a grouped bar chart. Her initial specification produces a blank chart with axes. Upon inspecting the data flow graph,

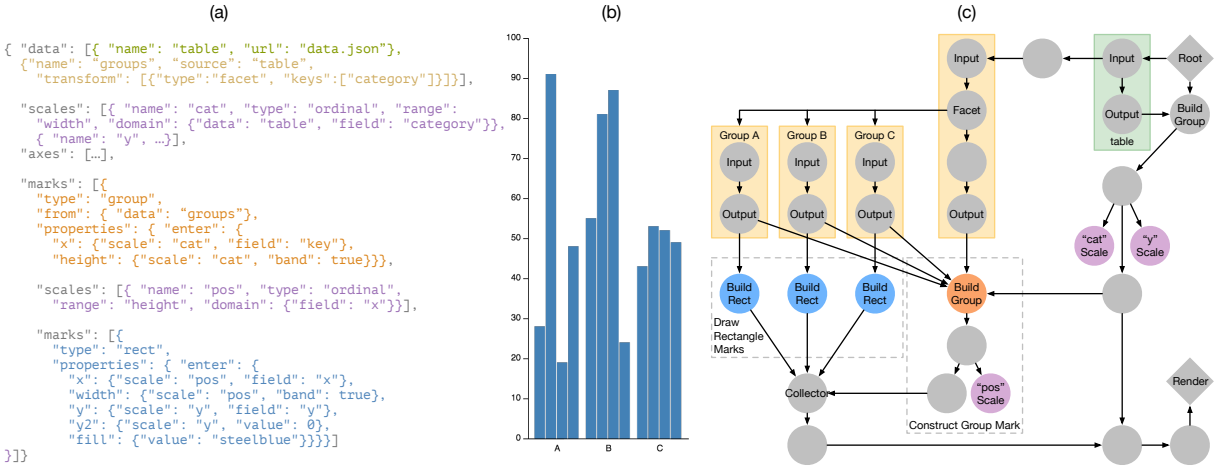

Figure 1: *The components of a Vega workflow with related elements colored; (a) the specification, (b) the output visualization, and (c) a simplified representation of the underlying data flow graph.*

she realizes that the rectangles are being drawn from the root of a hierarchical data source, not the faceted leaves. To resolve the error, she 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 than the specification alone (Figure 1a, c).

This graph represents the execution structure for rendering a Vega specification, but there are a number of questions as to the best way to use this structure to facilitate the developer's workflow. An accurate portrayal of the data flow graph contains intermediary nodes and edges, such as the `Collector` and unlabeled nodes of Figure 1, that are artifacts of Vega's low-level implementation and optimizations. For example, the `Build Group` node that constructs the group mark is not directly connected to the nodes drawing the rectangle marks, but all nodes are instead routed through a `Collector`. This relationship makes it hard to infer how the rectangle marks are related to the group mark and faceted data source. As a result, an initial challenge is identifying and abstracting visualizations of the data flow graph to be useful to developers.

The current Vega workflow consists of first writing a specification, and then handing it off to the Vega library to parse and render the final visualization. This deferred evaluation creates a lag between authoring a visualization and debugging the resultant output, making it hard to pinpoint the source of errors. A visualization of the data flow graph can tighten this feedback loop, enabling a more iterative design process. Brushing and linking allows developers to jointly inspect the specification, visualization and data flow graph, but challenges such as the behavior of interactive visualizations and hierarchical specifications makes identifying correlated elements difficult. Additional techniques, such as transient overlaid guides [SH14], will be necessary to account for these complications.

However, visualizing the data flow graph alone does not provide insight into how data propagates to the resultant visualization. Such insight would be essential if changes to the data are not realized in the visualization as expected. Allowing developers to step through the data propagation cycle enables rapid identification of how data flows through and is changed by each operator in the graph. By identifying how data changes at each point, the developer can locate the node at which her expectations diverge from the actual behavior. This step-by-step exploration of the data propagation cycle would also enable the developer to examine how branches of the data flow graph are dynamically added to support changes in the data. For example, new nodes would be added to the data flow graph of Figure 1c to support the addition of a fourth category in the data source.

3. Conclusion

The proposed techniques can help bridge the gulf of evaluation that is introduced by decoupling specification and execution. By presenting and augmenting the data flow graph, developers can jointly inspect the specification, visualization, data flow graph, and streaming data to iteratively develop visualizations. In general, visual representations of program states can provide developers with the context necessary to better interpret and interact with their code. Surfacing the program state allows developers to adjust their mental model alongside the program execution and can therefore enable more effective debugging practices by limiting this separation.

References

[BOH11] BOSTOCK M., OGIEVETSKY V., HEER J.: D3: Data-driven documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2011). URL: http://idl.cs.washington.edu/papers/d3. 1

[Guo13] GUO P. J.: Online Python Tutor: Embeddable web-based program visualization for CS education. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2013), SIGCSE '13, ACM, pp. 579–584. URL: http://doi.acm.org/10.1145/2445196.2445368, doi:10.1145/2445196.2445368. 1

[HB10] HEER J., BOSTOCK M.: Declarative language design for interactive visualization. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2010). URL: http://idl.cs.washington.edu/papers/protovis-design. 1

[LBM14] LIEBER T., BRANDT J. R., MILLER R. C.: Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2014), CHI '14, ACM, pp. 2481–2490. URL: http://doi.acm.org/10.1145/2556288.2557409, doi:10.1145/2556288.2557409. 1

[SH14] SATYANARAYAN A., HEER J.: Lyra: An interactive visualization design environment. *Computer Graphics Forum (Proc. EuroVis)* (2014). URL: http://idl.cs.washington.edu/papers/lyra. 2

[SWH14] SATYANARAYAN A., WONGSUPHASAWAT K., HEER J.: Declarative interaction design for data visualization. In *ACM User Interface Software & Technology (UIST)* (2014). URL: http://idl.cs.washington.edu/papers/reactive-vega. 1

[Veg14] Vega: A Visualization Grammar. http://trifacta.github.io/vega, April 2014. 1