Designing Interaction Support for Mascot Data Visualization Grammar

Charlie Lu
University of Maryland College Park
clue@umd.edu

Abstract

Interactive data visualizations act as a bridge between raw data and meaningful insights in a way that's not possible with static designs. Mascot, short for Manipulable Semantic Components, is a data visualization grammar designed to support the generation of expressive charts and currently does not have support for interactive visualizations. In this paper we aim to explore the design for the framework that will enable interaction within Mascot visualizations.

1 Introduction

Data visualization grammars exist to empower designers by providing a structured framework for creating visualizations. Declarative grammars such as Vega[1], ggplot2[9], and D3[4] allow users to express "what" they want to visualize without delving into the intricacies of "how" to achieve it[5]. Unlike some traditional declarative languages that primarily focus on specifying the "what" (e.g., what visual elements to show and how they relate), Mascot emerges as a high-level grammar that provides abstractions for both the structure and the procedure involved in constructing a chart[2].

This dual emphasis allows users not only to define the visual elements and their relationships but also to have a measure of control over the procedural aspects of how the chart is created. By specifying the steps involved in the construction process, Mascot provides users with more granular control and flexibility in shaping the final visualization. The emphasis on both structure and procedure is a more programmatic approach, enabling users to influence not just the appearance but also the underlying relationships of chart objects.

Mascot aims to be able to support interactive visualizations in addition to its static charts. To this end, we intend to be able to cover a whole range of user intents as in coverage over Yi et al.'s interaction taxonomy: select, explore, reconfigure, encode, abstract/elaborate, filter, and connect[10]. The subsequent sections will explore the pivotal role of dependency graphs in Mascot's architecture. These visual representations not only capture the relationships within a chart but also lay the groundwork for understanding the dynamic behaviors that support user interactions.

2 Related Work

Mascot draws on prior work in data visualization grammars. There are several data visualization tools that we draw from. Vega, Vega-lite, and D3 are a few that Mascot examined. D3 is a powerful foundational library of which, Mascot, Vega[1] and by extension Vega-lite[6] are built on. It offers a low-level approach which allows it to become a foundational building block of higher-level chart libraries[4].

Vega and by extension Vega-lite provide a higher-level abstraction than D3. The method in which Vega handles chart construction is through a JSON specification from which Vega's JavaScript runtime parses to generate their view models[7].

2.1 Vega Dataflow Graph

Vega employs a dataflow graph as a foundational structure to define the relationships between visual elements[8]. The Vega dataflow graph, created from a declarative specification, represents the flow of data and transformations, capturing the essence of how data is taken, processed, and eventually mapped to visual properties. This architecture enables a clear separation between the specification of visual elements and the underlying data transformations, facilitating a more concise and expressive method for creating com-

plex visualizations. The dataflow graph in Vega serves as a key mechanism for orchestrating the dynamic aspects of visualizations, providing a powerful foundation for building intricate and interactive data-driven graphics[7].

3 Overview of Mascot architecture

Mascot is a semantic-components-based visualization grammar that offers a high-level abstraction for the structure of a chart and the procedure to construct and modify its structure[3]. The procedure is defined by a series of operations e.g., repeat, divide, and densify. In this way, it is intended to support the creation of charts from a graphics-centric perspective.

Every visualization in Mascot are composed from four semantic components: visual object, layout, encoding & scale, and constraint. These components collectively define the structure and appearance of charts generated through Mascot.js.

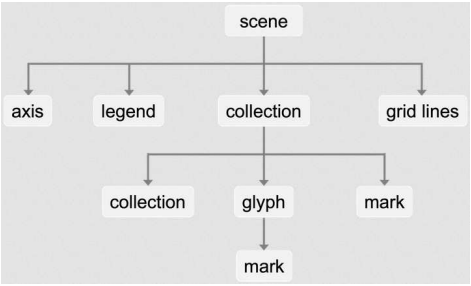

Figure 1: Scene graph showing visual objects hierarchy

3.1 Visual Object

A visual object in Mascot encompasses basic graphical elements such as marks, including geometric shapes like circles or polygons, text elements, and images. These serve as the foundational building blocks for creating visualizations.

- **Marks** are basic graphical elements in Mascot visualizations. Marks can be a geometric shape, text element, or image.
- **Collections** in Mascot group visual objects, each representing a distinct data item. For instance, a bar chart might consist of a collection of rectangle marks, with each rectangle symbolizing a specific month.
- **Glyphs** are groups of marks within Mascot that convey attributes of the same data item. In scenarios like a box-and-whiskers plot, glyphs may include multiple marks, each representing different attributes such as maximum, minimum, or the 75th percentile.
- **Reference Objects**, including axes and legends, provide contextual information essential for interpreting visualizations. They contribute to the overall understanding of how to read and interpret the chart.
- A **scene** in Mascot constitutes a self-contained view, comprising mark/glyph collections and reference objects. Scenes encapsulate specific aspects of a visualization and aid in organizing visual components.

3.2 Parametric Layout

Mascot employs parametric layouts to determine the positions of visual objects within a collection. Layout types such as grid, stack, packing, and treemap offer flexibility, allowing adjustments to layout parameters.

3.3 Encoding & Scale

Visual encoding specifies the mapping between a visual channel and a data field. For instance, the height of rectangles in a bar chart may encode sales values. The scale component determines how data values map to the properties of a visual channel.

3.4 Constraint

Constraints define spatial requirements for visual objects, influencing their relative arrangements. Examples include aligning objects to the left or maintaining specified distances between elements.

3.5 Scene Graph

The underlying mechanism behind how Mascot builds its charts is by employing a scene graph as a fundamental data structure to represent the hierarchical composition of visual elements in a data visualization. It organizes visual components, such as marks, glyphs, collections, and axes, into a tree-like structure that reflects their relationships and nesting within the visualization.

4 Dependency Graph

While the scene graph provides a hierarchical structure for organizing components, it does not explicitly capture the dynamic relationships and dependencies needed to have support for interactive visualizations. As such, we recognize the importance of maintaining a clear representation of dependencies and interactions among these elements. We turn our attention to the design of the dependency dataflow graph. This graph extends beyond the static arrangement of visual elements and focuses on modeling the dynamic interac-

tions and data dependencies that drive the construction and updates of a visualization.

In the Mascot data visualization system, the dependency graph serves as a pivotal mechanism for capturing the intricate relationships between various elements, allowing for dynamic updates and maintaining the coherence of visualizations. This graph consists of nodes representing scene element properties, layout builders, scale builders, and width and height encoders. Edges between these nodes signify dependencies, outlining the flow of influence from one property to another as updated values propagate through a scene.

4.1 Node Types

Within the Mascot dependency graph, nodes are broadly classified into two main categories

4.1.1 Scene Element Properties

These nodes reflect properties of elements in the scene. For instance, nodes representing mark properties attributes like height, width, and x/y position. These properties are fundamental to defining the visual appearance and positioning of graphical elements within a chart.

4.1.2 Operators

Operator nodes function as crucial intermediaries within the dependency graph of a chart. These nodes process changes, serving as pivotal components that facilitate the arrangement and behavior of visual elements. In essence, they act as intermediaries between specific scene element property nodes, ensuring that changes propagate seamlessly through the dependency graph, reflecting the overall structure and behavior of the visualization. These nodes can be divided into two categories:

Builders: Builder Nodes encompass a diverse category constituting nodes that receive input and produce output. These nodes encapsulate various functionalities, ranging from algorithms responsible for determining the spatial arrangement of visual elements to encoders and scales. For instance, a figure (Figure X) illustrates how a builder node orchestrates the layout of visual elements within a chart, providing a tangible example of its role in the broader context of the dependency graph.

Constraints: These nodes are not connected by directed edges. Instead, these nodes are reserved for any signals that might propagate to all connected nodes. They act as conduits for signals that synchronize specific attributes among connected nodes. For instance, an equality node passes along a change to all connected nodes, ensuring their values remain equal.

4.2 Edge Types

Edges between nodes indicate the flow of influence within a scene. When a property or parameter undergoes a change, it initiates a cascading effect on its downstream dependents. For instance, modifying the scale of a color encoding can lead to updates in the color representation of marks, influencing the visual appearance of the entire chart. In the Mascot dependency graph that we are developing, we define two different types of edges.

Directed Edges for Signal Propagation: These edges signify the directional influence of changes within the dependency graph. When a property or parameter undergoes modification—such as adjusting the scale of an x/y axis or altering a layout—the directed edges ensure a cascading effect. Downstream nodes, dependent on the changed property, dynamically update to maintain coherence within the visualization.

Equality Edges for Shared Properties: In addition to directed edges, equality edges establish undirected connections among nodes to facilitate property sharing. In scenarios where visual elements, like bars in a multibar chart, share common attributes (e.g., left alignment), equality edges ensure that a change in one component propagates equally to others. This bidirectional linkage promotes consistency in properties across related visual elements.

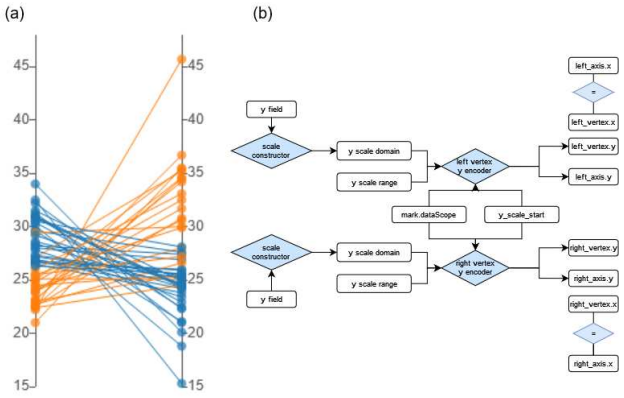

Figure 2: (a) A slope chart and (b) the corresponding section of the dependency graph

4.3 Dependency Graph Example

In order to better understand how these node and edges work together to form a dependency graph, we'll illustrate an application of the dependency graph in Mascot. We'll examine a common visualization type, the slope chart. A slope chart typically displays the change in values between two points. Let's consider the simple slope chart in Figure 2.

4.3.1 Slope Chart

In the corresponding dependency graph for this slope chart in Figure 2(a), various nodes represent key elements such as position, size, and color. Each slope is considered a mark and consists of two vertices that lie on either axis. This dependency graph captures the relationships between these elements, enabling dynamic updates when underlying data changes. Figure 2 zooms in on the section of the graph that reflects the positioning of the axes and marks. For example, a change in the y field for either axis eventually propagates to the y-values of the corresponding vertices by triggering a cascading update through the dependency graph. In addition, the example illustrates how undirected edges, specifically in the form of an equality constraint, play a crucial role. As the vertices lie on the axis, an update to the x position of one axis similarly updates the x positions of the corresponding vertices.

Exploring specific examples and their corresponding dependency graphs provides valuable insights into the broader principles of how visualizations are constructed within the Mascot framework. By delving into various chart types, our team gains a nuanced understanding of the dependencies that govern their dynamic behavior. These examples serve as practical guides, illustrating how changes in data fields propagate through the dependency graph, influencing the visual elements' attributes. As we examine diverse chart structures, from scatter plots to bar charts, we established a foundational knowledge base for creating robust and adaptable dependency graphs. This proactive exploration of examples equips our team with the foresight needed to implement effective and efficient dependency graphs, laying the groundwork for the continued development of the Mascot visualization framework.

5 Conclusion and Future Work

Creating a robust grammar depends on creating a solid foundation on which to build upon. Mascot is not aiming to define a few common interactions for most plots but to be able to allow the users to define interactions themselves. With a robust dependency graph understanding and definition we can allow that to happen.

6 Acknowledgments

I'd like to thank Professor Zhicheng Liu and Shubham Karanjekar for their assistance.

References

[1] Vega: A visualization grammar. http://trifacta.github.io/vega, April 2014.

[2] Mascot.js - manipulable semantic components in data visualization. https://mascot-vis.github.io/, 2021.

[3] Mascot.js - manipulable semantic components in data visualization: Overview of component model. https://mascot-vis.github.io/tutorials/vom/ , 2021.

[4] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D^3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011.

[5] Jeffrey Heer and Michael Bostock. Declarative language design for interactive visualization. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1149–1156, nov 2010.

[6] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. Vega-lite: A grammar of interactive graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):341–350, 2017.

[7] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE Transactions on Visualization & Computer Graphics (Proc. IEEE InfoVis)*, 2016.

[8] Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. Declarative interaction design for data visualization. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, page 669–678, New York, NY, USA, 2014. Association for Computing Machinery.

[9] Hadley Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer, 2009.

[10] Ji Soo Yi, Youn ah Kang, John Stasko, and J.A. Jacko. Toward a deeper understanding of the role of interaction in information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1224–1231, 2007.