Declarative Interactive Visualization in Mascot: Dependency Graph Architecture

Shubham Vasantrao Karanjekar
University of Maryland, College Park. MD
shubhk@umd.edu

Abstract

Several well-known tools for declarative data visualization, such as D3, Vega, and Mascot (version 1.0), allow users to describe visualizations using high-level specifications or syntax, removing the need for explicit programming at each step. These tools excel in creating static visualizations but frequently lack straightforward support for flexible declarative interactions, especially in d3. Oftentimes, users must manually incorporate event listeners to detect actions and implement custom logic. This paper presents an architecture inspired by Streaming Databases and Reactive Vega [1], leveraging a Dependency Graph. This architecture not only captures user inputs but also dynamically generates a scene and a dependency graph (DataFlow Graph) at runtime. These graphs react to appropriate input changes, enabling efficient rerendering of visualizations.

1 Introduction

Mascot[2], which stands for **Ma**nipulable **S**emantic **Co**mponen**t**s, presents a declarative data visualization grammar geared towards creating expressive charts. It offers high-level abstractions for both chart structure and construction procedures. However, like many other popular declarative data visualization frameworks, Mascot lacks essential support for declarative interaction. Users often find themselves needing to code interactions manually, introducing potential complexities.

This paper addresses this gap by introducing a simplistic, structural, and declarative approach based on functional reactive programming principles into Mascot. Drawing inspiration from Reactive Vega[1], our work aims to extend interaction support beyond just runtime interactions. We introduce design-time interactions, allowing users to interact not only with complete charts but also with individual chart elements such as axes, legends, and data points.

Our approach begins by dynamically generating a memory-based dependency graph, typically a Directed Acyclic Graph (DAG), though occasionally including cycles. This graph delineates various operators and variables, connected by edges that describe their relationships.

In striving to support interactive visualizations alongside static charts, Mascot aims to encompass all seven user intents outlined in Yi et al.'s interaction taxonomy[5]. Through this comprehensive coverage, Mascot seeks to empower users with versatile and intuitive interaction capabilities, enhancing the usability and effectiveness of declarative data visualizations. As we will delve into in Section 4, Mascot has introduced support for two types of interactions: *Design Time and Runtime*. *Design Time interactions* involve users acting on specific semantic components (such as Marks, axes, etc.) that trigger the data flow within the dependency graph. On the other hand, *Runtime interactions* encompass actions like zooming and brushing, which (primarily) affect the entire scene.

2 Related Work

Drawing inspiration from various tools and frameworks such as Vega, Vega-lite[4], Reactive Vega, D3[7], Database Streaming Dataflow graphs, Mascot builds upon existing work in the field of data visualization grammar. Its low-level approach allows it to serve as a fundamental building block for higher-level chart libraries.

Reactive Vega[1] introduces a novel system architecture that addresses declarative visual and interaction design comprehensively. It constructs a dataflow graph from a single declarative specification, treating input data, scene graph elements, and interaction events as first-class streaming data sources. This architecture enables the creation of expressive interactive visualizations, capable of handling time-varying data dynamically. While Reactive Vega excels in runtime interactions, it currently lacks support for design-time interactions.

SkyBlue[9] presents an incremental constraint solver (another name for dependency graph) that utilizes local propagation to manage a set of constraints efficiently. It maintains constraints individually, allowing for seamless addition and removal while ensuring consistency.

Bluefish extends UI architectures to support overlapping perceptual relations, offering a framework for authoring graphic representations across diverse domains. Bluefish graphics are instantiated as relational scenegraphs, preserving the compositional and abstractional affordances of traditional UI frameworks [10]. However, Bluefish currently lacks support for interactions.

3 Dependency Graph architecture in Mascot

3.1 Mascot Grammar

Mascot [2] offers a comprehensive framework for chart creation by providing high-level abstractions for both chart structure and construction procedures. In terms of structure, it employs semantic components like marks, glyphs, collections, layouts, and encodings to define the foundational elements of a chart. The construction process involves a series of operations such as repeat, divide, and densify, which manipulate these semantic components to generate the desired visualization. Central to Mascot's approach is its focus on graphical objects as primary entities, allowing users to manipulate their properties, transformations, layouts, and constraints with ease. Its procedural syntax facilitates the application of operations in a step-by-step manner, enabling users to inspect visualization components at any stage of development. Additionally, Mascot offers flexibility in rendering, decoupling visualization logic from rendering mechanisms, allowing users to choose between SVG or WebGL renderers based on their preferences or requirements.

3.2 Dependency Graph Building Blocks

In this advanced iteration of Mascot, the dependency graph takes on a crucial role in capturing the intricate relationships between various elements, allowing for dynamic updates and maintaining the consistency of visualizations. As described by Lu et al.[8], they outline a dependency graph grammar consisting of nodes that represent both variables and operators. These operator nodes serve as essential intermediaries within the dependency graph, governing the behavior of visual elements by handling changes. Acting as the core logic for processing a set of inputs and generating a variable output, they serve as bridges between specific property nodes, ensuring that changes flow smoothly throughout the graph, thus reflecting the overall structure and behavior of the visualization. Meanwhile, variable nodes act as the primary sources of data and trigger points for data changes, initiating the propagation of flow within the dependency graph.

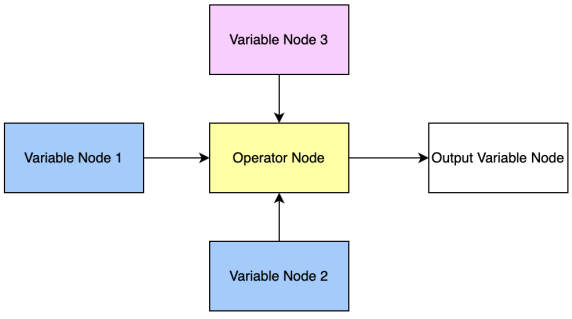


Figure 1: A Sample Dependency Graph

Figure 1 illustrates a basic dependency graph structure comprising three input variables, with each variable type represented by a different color. At the center lies an operator node, which takes in all the inputs and gener-

ates an output based on them. It's important to note that an operator can handle anywhere from one to multiple inputs, which can be of varying types.

3.2.1 Operators and Variables

We'll now delve into the various types of operators and variables present in the graph, discussing their significance and roles within the system.

Each node within the graph is assigned a unique identifier. A **Variable** node encompasses details regarding its incoming and outgoing edges, along with specifying the type of edge it represents (directed or undirected). Various types of variable nodes exist, each serving distinct purposes. For instance, a *ScaleVar* extends from its parent and additionally stores information regarding the encoding scale of a visual element. Similarly, a *PropertyVar* includes details about the type of property it represents and the corresponding Mascot's basic element, such as mark, glyph, group, or axis. Additionally, a *DatascopeVar* holds all the necessary source data required for rendering the graph, and *CondEncodingVar*, holding the encodings based on events such as click and/or brush. The system allows for the creation of additional variable node types based on specific needs and requirements.

An operator node comprises information about all input and output variable nodes, along with a run method that dynamically triggers execution at runtime, facilitating scene formation or responding to event changes. Similar to variables, there are various types of operators available. For example, *RangeBuilder* and *DomainBuilder*, as suggested by name, aid in constructing the domain and range for a visualization, triggered by changes in the datascope information. These operators are designed to intuitively support streaming data changes. Furthermore, operators like *Property Encoders*, *Evaluators*, and *AxisEncoders* are utilized to encode various aspects such as properties (x, y, fillColor), bounds, range, domain of marks on the screen, and axis information, respectively.

4 Declarative Interaction in Mascot

As mentioned earlier, Mascot dynamically constructs a dependency graph during runtime based on the specified parameters for visualization creation. Currently, Mascot supports two types of interactions: design-time interactions and runtime interactions.

4.1 Design-Time Interactions

These interactions involve user actions on the fundamental building blocks of a visualization. For example, if a user wants to change the starting position of a graph, they can simply grab any axis and move it on the screen. Similarly, altering the encoding of data points, such as changing the fill color of circles in a scatter plot, constitutes a design-time interaction. Consider the scenario where a user wishes to adjust the starting position of the graph. Mascot's built-in event listeners automatically capture this action and trigger the appropriate node in the dependency graph—in this case, the *RangeStartVar* of the *RangeBuilder*. This triggers a cascading effect, initiating a propagation flow throughout the dependency graph until all dependencies are resolved. Referring to the figure 2, a change in the *RangeStartVar* triggers the *RangeBuilder* operator, which executes its business logic based on the input change and outputs a *RangeVar*. Subsequently, the change in the *RangeVar* triggers the corresponding *Encoder* operator based on the input channels, and this cycle continues. In this example, the dependency graph forms a simple Directed Acyclic Graph (DAG), allowing for straightforward topological sorting to capture the order and relationship of the trigger flow.

4.2 Run-Time Interactions

These interactions operate on the same logic as design-time interactions but encompass different paths and occur during runtime.

Runtime interactions in Mascot are managed through the "activate" API within the scene class, utilizing a set of predefined JSON-based properties. These properties, comprising *target*, *criteria*, *effect*, and *event* fields, drive the interaction behavior.

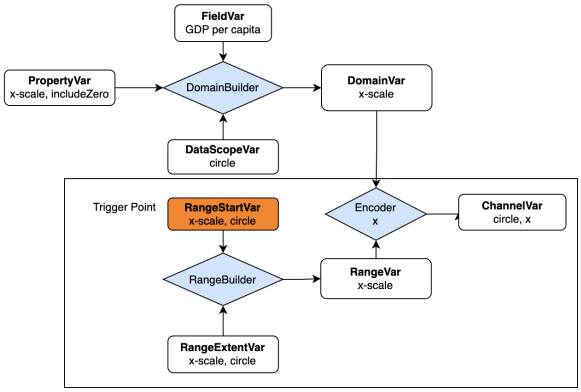


Figure 2: Design Time Interaction Flow in Dependency Graph of Mascot

To begin, a specific target, such as a mark, is identified to undergo visual changes triggered by user interactions. Parameters like criteria and effect are then specified. The criteria parameter dictates how the visual elements, respond to user actions, and a logical expression is constructed during runtime based on JSON specification. The effect parameter determines how the appearance of elements changes based on the evaluated criteria.

These declarations dynamically contribute to constructing a portion of the dependency graph at runtime. Some nodes may reuse existing structures, while others generate new ones using *PredicateVar* variable nodes and *PredicateEncoder* operators. The *PredicateVar* encompasses various types, such as *IntervalPredicate*, *ListPredicate*, and *PointPredicate*, tailored to specific use cases. For instance, a *PointPredicate* might suffice for a simple click event in a scatter plot, whereas an *IntervalPredicate* would be more suitable for defining a selected region in a brush event. The *ConditionalEncoding* class encapsulates the logic pertaining to the target element, predicates, and effects in Mascot.

The event property specifies a standard HTML event and is handled by the respective Mascot renderer functions, ensuring a straightforward and scalable approach to managing runtime interactions.

Consider a scenario where we have a scatter plot visualization 5(a), and upon clicking any circle mark, all corresponding circles with the same color encoding should become "active," while others fade out 5(b). The color attribute is encoded by the "Continent" field of the dataset.

To achieve this interaction, we utilize the Mascot scene's "activate" API. Here's how we specify the interaction 3:

```
scn.activate(circle, {
    criteria: { attribute: 'Continent', type: 'point' },
    effect: {
        fillColor: { false: "#eee" },
        opacity: { false: 0.3 }
    },
    event: "click"
});
```

Figure 3: Example Specification for the Mascot's Activate API

In this setup, clicking on a circle mark triggers the defined criteria, which evaluates the "Continent" attribute of the dataset to determine relevant circles. The effect parameter specifies that circles whose continent attribute does not match the clicked circle's continent value should have their fillColor set to #eee, effectively fading them out. Conversely, circles with the same continent attribute as the clicked circle remain active, maintaining their appearance. To expand further, the colors of the circles are determined by both data binding "Continent" and conditional encodings. Additionally, the dataflow path is determined by the captured event, and a suitable ConditionalEncoding Variable is selected at runtime, containing all the information regarding predicates and rules.

This interaction enhances user engagement by visually highlighting circles of interest while de-emphasizing others, contributing to a clearer understanding of the dataset's distribution and patterns within the scatter plot visualization. Figure 4 illustrates these dependencies in the form of a dependency graph. All computations related to color encoding are managed by the Encoder operator.

5 Conclusions

The ultimate objective was to furnish a declarative and functional reactive method for incorporating interactions into both static chart

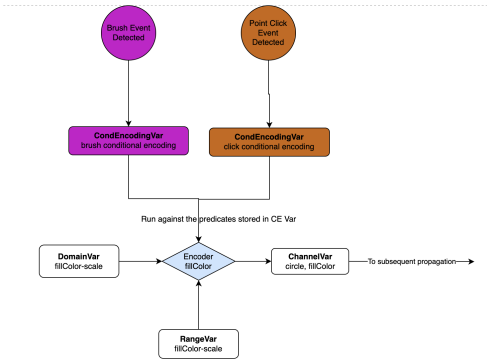
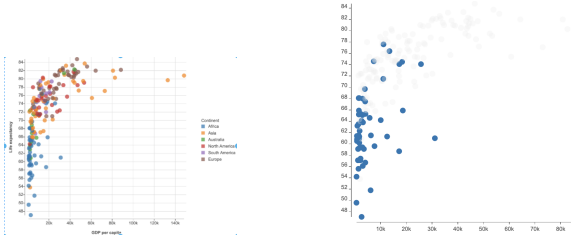


Figure 4: Dependency Graph flow post Click Event in Scatter Plot



(a) Scene with regular scatter plot

(b) Screen after click interaction

Figure 5: Mascot Run Time Interactions Effects

design and runtime usage, and the Dependency Graph model of Mascot emerged as the solution. This approach not only enhances user accessibility but also ensures optimal performance. As part of future endeavors, there's potential to extend Mascot's capabilities to encompass streaming data and animation functionalities, thereby enhancing its versatility and applicability in various visualization scenarios.

6 Acknowledgements

I am deeply grateful to Dr. Professor Leo Zhicheng Liu for his invaluable guidance and mentorship throughout this work.

References

[1] 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 Info-Vis), 2016.

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

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

[4] 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.

[5] 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.

[6] 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, pages 669–678, New York, NY, USA, 2014. Association for Computing Machinery.

[7] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3 data-driven documents. IEEE Transactions on Visualization and Computer Graphics, 17(12):2301–2309, 2011.

[8] Charlie Lu, Designing Interaction Support for Mascot Data Visualization Grammar

[9] Michael Sannella, SkyBlue: A Multi-Way Local Propagation Constraint Solver for User Interface Construction

[10] Josh Pollock, Catherine Mei, Grace Huang, Daniel Jackson, Arvind Satyanarayan, Bluefish: A Relational Framework for Graphic Representations