Declarative Interaction Design for Data Visualization

Arvind Satyanarayan
Stanford University
arvindsatya@cs.stanford.edu

Kanit Wongsuphasawat
University of Washington
kanitw@uw.edu

Jeffrey Heer
University of Washington
jheer@uw.edu

ABSTRACT

Declarative visualization grammars can accelerate development, facilitate retargeting across platforms, and allow language-level optimizations. However, existing declarative visualization languages are primarily concerned with visual encoding, and rely on imperative event handlers for interactive behaviors. In response, we introduce a model of declarative interaction design for data visualizations. Adopting methods from reactive programming, we model low-level events as composable data *streams* from which we form higher-level semantic *signals*. Signals feed *predicates* and *scale inversions*, which allow us to generalize interactive selections at the level of item geometry (pixels) into interactive queries over the data domain. *Production rules* then use these queries to manipulate the visualization's appearance. To facilitate reuse and sharing, these constructs can be encapsulated as named *interactors*: standalone, purely declarative specifications of interaction techniques. We assess our model's feasibility and expressivity by instantiating it with extensions to the Vega visualization grammar. Through a diverse range of examples, we demonstrate coverage over an established taxonomy of visualization interaction techniques.

Author Keywords

Visualization; interaction design; toolkits; declarative design.

ACM Classification Keywords

D.2.2 Software Engineering: Design Tools and Techniques; I.3.6 Methodology and Techniques: Interaction Techniques

INTRODUCTION

Declarative languages decouple specification (the *what*) from execution (the *how*). Popular in domains ranging from web design (e.g., HTML/CSS) to database queries (e.g., SQL), declarative languages are now widely used to design custom data visualizations as well. Designers describe mappings between data values and properties of graphical primitives; a language runtime then determines appropriate control flows for data processing, rendering, and animation [5, 6, 40]. This decoupling lets users focus on visual encoding decisions, leaving the runtime to unobtrusively optimize processing [17]. Declarative design also simplifies retargeting, for instance allowing a visualization specification to be rendered in a browser using HTML5 Canvas or Scalable Vector Graphics (SVG), or as a static image for print-based media [36].

However, these languages provide little support for declarative design of *interactions*. Some tools [5, 6] offer a predefined palette of interaction techniques (brushing, zooming, etc.) that can be declaratively applied. However, these "interactor typologies" artificially restrict designers' options and typically support customization of only a limited set of parameters. For custom interactions, designers must instead author *imperative* event handling callbacks. These callbacks undo the benefits of declarative design by exposing execution details, forcing users to manually maintain state [8] and coordinate interleavings — a complex and error-prone task colloquially referred to as "callback hell" [12].

In this paper, we introduce a model of declarative interaction design for data visualizations with expressiveness comparable to imperative approaches. We first model interactive input events as composable *streams* of data. Input event streams are treated as first-class data sets, and thus are subject to the full range of applicable data processing operators, including filtering, aggregation and summary statistics. Interactions can also be composed into arbitrary reactive expressions [38] called *signals*. When a new event enters a stream, it propagates to dependent signals and the expressions are automatically re-evaluated. This approach, modeled on Functional Reactive Programming (FRP) [3], defers the complexity of coordinating event-driven state changes to the language.

By default, interactions occur at the visual (pixel) level. However, with visualizations, this is often insufficient. Instead, to maximize reuse and expressivity, interactions must be lifted to the data level [16]. Under our model, signals can be used to construct intensional and extensional *predicates* that define membership in an interactive selection. By passing signals through *scale inversions* [11, 16] that map visual values to corresponding data values, predicates can also express dynamic queries over data. Predicates can then be used to modify visual encodings using declarative *production rules*.

Our model also facilitates reusing interactions both within and across designs. Interaction specifications (streams, signals, predicates, and rules) can be encapsulated as named *interactors*: standalone declarative specifications that provide a file format for interaction techniques. Moreover, streams and signals are named to decouple low-level event processing from application-level semantic events. Downstream interaction logic is defined in terms of these named, semantic properties and not low-level input events. For example, when combining two conflicting interactions or to retarget interactions

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
UIST '14, October 05 - 08 2014, Honolulu, HI, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3069-5/14/10 $15.00.
http://dx.doi.org/10.1145/2642918.2647360

across form factors, we can simply rebind signals to different low-level input events without any downstream modification.

We instantiate this model through reactive extensions to Vega, a JSON-based visualization grammar [36]; an example Vega visualization is shown in Figure 1. Although our model could be implemented within other frameworks, Vega's JSON environment allows us to assess the extent to which our model enables *declarative* interaction design, and evaluate its expressivity. We construct a diverse range of examples which demonstrate substantial coverage of an existing taxonomy of interaction techniques for data visualization [42], including brushing & linking, panning, zooming, and filtering with control widgets. These examples show that, through composition, reuse, and repurposing of interaction techniques, our model promotes the accretive design of richer interactions.

RELATED WORK

Our declarative interaction model extends prior work on visualization systems, reactive programming and UI toolkits.

Interaction Support in Visualization Toolkits

Interaction is an important component of effective data visualization [25, 32], and most existing tools support a set of common interaction techniques. Prefuse [18] provides controls for focus, hover, drag, and tooltips, along with a predicate language to express dynamic queries. Improvise [39] introduces *live properties* — active variables that parameterize a visualization and can be bound to control widgets. Live properties take part in *coordinated queries* for linked brushing or synchronized scrolling across multiple views.

Our model shares some conceptual underpinnings with these systems. Improvise's live properties provide a basic form of reactivity, while coordinated queries and prefuse's predicates lift interactions to the data domain. However, both systems take a monolithic approach to interaction design. Interactors must be defined within a rigid class hierarchy, and subclassed to be customized or composed. This approach complicates interaction design and reduces flexibility, requiring entirely new imperative code to modify input event handling or to target new platforms (e.g., from mouse to touch input).

Declarative tools such as Protovis [5] and D3 [6] similarly offer a palette of standard interaction techniques, with black-boxed event processing and limited customization via exposed parameters. Custom interaction designs require imperative event handling callbacks, imposing different specification styles on visual design and interaction design [4]. These callbacks can stymie the benefits of declarative design by exposing execution details and requiring increased development effort. Other declarative models, such as Wilkinson's Grammar of Graphics [41] and Wickham's ggplot2 [40] do not include support for interaction.

Stencil [9] is a visualization language that models data using streaming semantics. When a data value changes, it is automatically propagated to visual glyphs, and the visualization re-renders. Stencil's authors note that user input can be modeled as a data stream, but they do not close the loop and extend their language to support interaction design. Stencil lacks constructs necessary to generate interactive selections and generalize them to dynamic queries [16].

Similarly, Quadrigram [33] (née Impure [30]) constructs visualization applications using dataflow semantics. However, the visualization components themselves are "black boxed" into a chart typology with built-in selection interactions. As it does not expose raw input events nor consider user interaction as a data source, Quadrigram's expressivity is restricted.

Constraint Programming

Imperative event callbacks often present development hurdles [28]. Callbacks registered outside the visualization specification allow users to externally manipulate visualization state and manually update dependencies [8]. This approach breaks encapsulation and necessitates the introduction of side-effects: external calls to maintain state [7]. Callback execution order can also be unpredictable, requiring users to coordinate interleaved calls [12]. As a result, callbacks can make it difficult to reason about the current state of the visualization, which is now the product of both the original specification and the side-effects of interaction callbacks.

One alternative to callback-oriented imperative programming is constraint-based specification. Systems like Gilt [28] and μconstraints [21] implement one-way constraints: when the value of a constrained interface is modified, its dependents are automatically affected. Other systems, like Cassowary [2], implement two-way constraints using more complex constraint solvers. Recently, ConstraintJS [29] and Bret Victor's prototype for drawing data visualizations [37] have shown that constraint programming is suitable for authoring web applications and data visualizations, respectively.

While constraint systems remove many of the obstacles introduced by callbacks, they do not involve general consideration of event handling, a crucial element for interaction design. In fact, the authors of ConstraintJS intend for their system to complement event architectures [29].

Functional Reactive Programming

Functional Reactive Programming (FRP) [13] formalizes semantics similar to one-way constraints. Drawing from dataflow programming [24], FRP recasts mutable states as

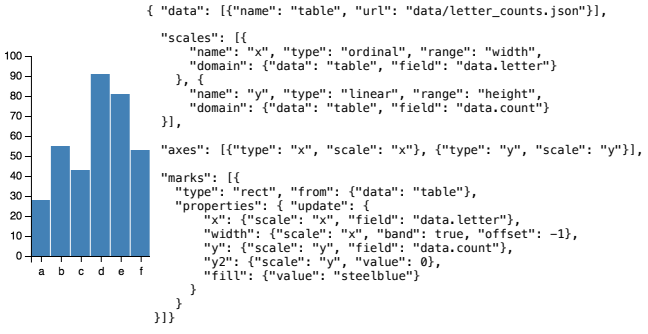

Figure 1. A JSON specification for a bar chart, demonstrating Vega's [36] existing abstractions. Data is imported from a URL. Scales are defined to transform data values to visual values. Properties of graphical marks (in this case rectangles) are determined by scale mappings. Guides (here, axes) can be instantiated as well.

time-varying streams of values. Among FRP variants [3], we focus on Event-Driven FRP (E-FRP), in which values change only in response to discrete events [38]. E-FRP models value changes with two constructs: *streams*, which are potentially infinite sequences of events, and *signals*, defined as the most recent event from a stream. Signals are akin to variables and can be composed into expressions. Runtimes for reactive languages construct a dataflow dependency graph; when a new event fires, the event enters the appropriate streams and propagates to any dependent signals, for which any dependent expressions are automatically re- evaluated. Languages like Flapjax [26] and ELM [10] demonstrate how E-FRP semantics can be used to author interactive web applications.

However, existing instantiations of E-FRP lack primitives critical for developing interactive *data visualizations*. By default, they only consider interaction events over the visual or geometric space of a browser window. For data visualization, it is important to be able to generalize selections to the data domain [11, 16, 39]. Our model adopts the semantics of E-FRP and extends it with primitives necessary to perform interactions at both the visual and data levels.

Reusable Interactions

Myers et al.'s Garnet system introduces *Interactors* [27]: a scheme for encapsulating interactive behaviors that enables reuse by decoupling input events from application logic. Our notion of interactors extends this work. In addition to endowing interactors with reactive semantics, in our model interactors expose their constituent components (signals, predicates, and rules); a visualization can choose to use an entire interactor or chosen subsets. Compared to Garnet, whose interactors can only be generalized through provided parameters, our interactors allow more granular reuse and repurposing.

DECLARATIVE INTERACTION MODEL DESIGN

With our model, we contribute a declarative grammar of interaction design for data visualization. It extends and seamlessly integrates with existing grammars of graphics, offering a lower-level compositional approach to interaction design. As we later demonstrate, our model facilitates development, enables richer customization, and allows greater reuse of interaction techniques without an undue loss of expressivity.

Event Streams and Signals

Our model adopts the semantics of Event-Driven Functional Reactive Programming (E-FRP) [38]. Low-level input events (e.g., mouse events and keystrokes) are captured through *streams*, rather than event callbacks. Abstracting event handling as streams reduces the burden of combining and sequencing events — operations that would require callbacks to coordinate external states. To this end, we introduce a syntax for selecting and composing event streams, as shown in Figure 2. While prior work has formulated regex-based symbols for event selection [23], we instead draw inspiration from CSS selectors. As a result, our syntax operators are likely to be more familiar to visualization designers.

An event stream is denoted using an event name (e.g., `mousemove`), optionally prepended with a mark type

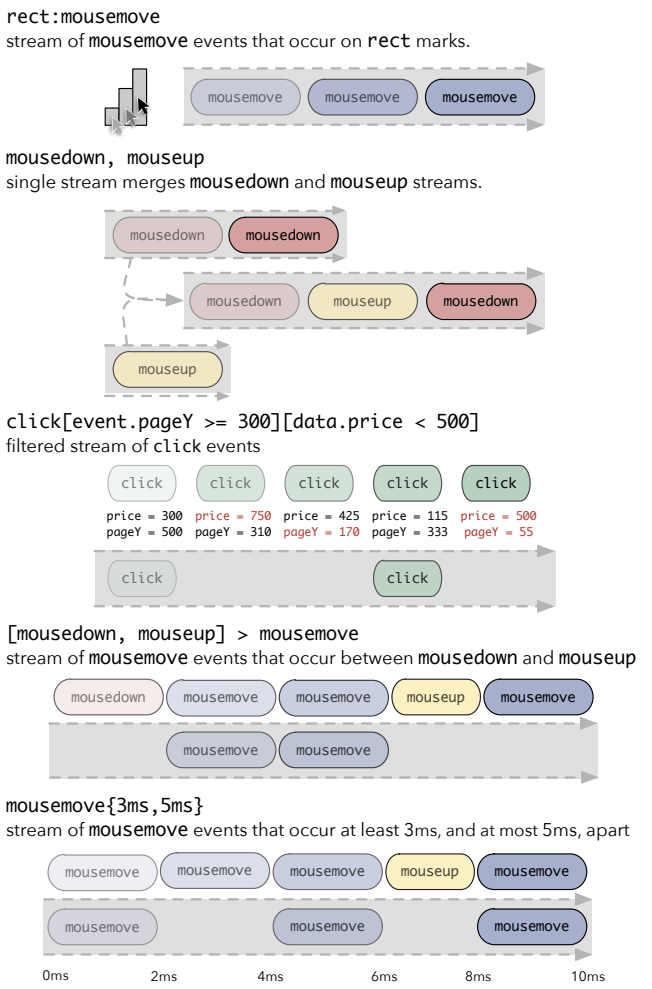

Figure 2. Event streams can be instantiated, composed, filtered, and sequenced using a syntax inspired by CSS3 selectors and regular expressions. This determines which events are captured by a particular stream (denoted by the dashed grey rectangles).

(e.g., `rect:`, or `symbol:`). The comma operator (`,`) merges streams to produce a single stream with interleaved events. Square brackets (`[]`) filter events based on their properties. When followed by the right-combinator (`>`), square brackets also define bounding events for the stream, serving as a "pre-filter" with the right-combinator selecting "children." Curly braces (`{}`) denote minimum and maximum time intervals between events. These operators are composable: `[mousedown, mousemove] > [keydown, keyup] > mousemove{5ms,10ms}` is a stream of `mousemove` events occurring at least 5ms apart and at most 10ms apart, between `keydown` and `keyup` events, which in turn occur between `mousedown` and `mouseup` events.

Critically, our model treats events as first-class data sources. Not only can data transformations (e.g., statistical calculations) be run over event streams, but events can also be composed into arbitrary reactive expressions as *signals*. By default, signals return the most recent event from a stream. However, by drawing from multiple event streams, they can

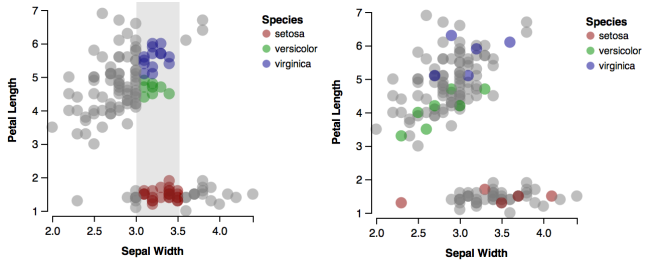

Figure 3. Points are highlighted using an intensional predicate to select points where `3.0 < data.sepalWidth < 3.5` (left) or with an extensional predicate to select points `#56, #110, #79, #95, #40, #120, ...` (right).

define finite-state machines — each stream triggering a transition between states, for example. By virtue of being modeled as data sources, signals can serve as input to visual encoding primitives (scales, guides, and marks) thereby endowing them with reactive semantics. When an event fires, it enters appropriate streams and is propagated to corresponding signals; signals are re-evaluated and dependent visual encodings re-calculated and re-rendered automatically.

Upon definition, streams and signals must be given unique names. These named entities are then used to define the rest of an interaction technique. This separation decouples input events from downstream application logic. Thus, an interaction can be triggered by a different set of events by simply rebinding stream and signal declarations. As we later demonstrate, rebinding is particularly useful for retargeting interactions or for combining otherwise conflicting interactions.

Predicates and Scale Inversion

Selection is a fundamental operation in interactive visualization design: subsequent operators are applied to selected items to manipulate them. For visual design, it can be sufficient to make a predetermined selection for example, "select all rectangles." With interaction design, however, selections are driven by user input. For instance, a user may brush over points of interest, or adjust a slider to filter data.

To express interactive selections, we introduce reactive *predicates*. *Intensional* predicates specify conditions over the properties of members while *extensional* predicates explicitly enumerate all members of the selection set [16]; this difference is shown in Figure 3. Predicates can be combined using logical `AND` or `OR`. They may be defined inline to toggle the properties of visual encodings but, if defined once at the top-level with a name and parameterized operands, can be reused.

Predicate operands are typically signals and, as signals are drawn from streams of input events, predicates express interactive selections at the visual level by default. However, pixel-level selection is often insufficient, as a single visualization may have multiple distinct visual spaces. In such cases, it is necessary to generalize an interactive selection into a query over the data domain [16]. For example, consider a simple overview+detail setup (Figure 4), in which the constituent plots have distinct coordinate spaces but shared data domains.

Scale functions are a critical first-class component in visualization design [41] as they transform data values into visual values such as pixels or colors. By applying an *inverted* scale function to predicate operands, we can lift a predicate to the data domain [11]. However, naïve application of scale inversion can breakdown when trying to express a range selection as a query. The semantics of quantitative scales allow a scale inversion of range extents to naturally transform pixel-level values to data values. The semantics of inverting discrete categorical or ordinal scales, however, are more complex.

To mask this complexity from the user, our model provides a special *range* predicate. This predicate calculates range extents in pixel space using either constants or signal values and, if a scale inversion is specified, produces a corresponding query: intensional for quantitative scales, extensional for categorical data. Figure 4 illustrates how range predicates and scale inversion allow brushing in the coordinate space of the overview plot to filter points displayed in the detail plot.

Production Rules

When predicates are applied directly to visual encoding properties, they serve as toggles. However, for more complex behaviors, our model provides *production rules*, an established design pattern [15] that we endow with reactive semantics. A rule defines the outcomes of evaluating a sequence of predicates (an `if-elseif-else` chain) to set property values. For example, a rule might set a mark's fill color to green if predicate `A` is true, use a scale-transformed data value if `B` is true, or otherwise set the color grey by default. Rules can be defined inline with visual encoding properties, or can be reused if defined at the top-level with a unique name and parameterized outcome branches.

User-Defined Functions

During our design process, we encountered visualizations in which interactions trigger custom data transforms. For exam-

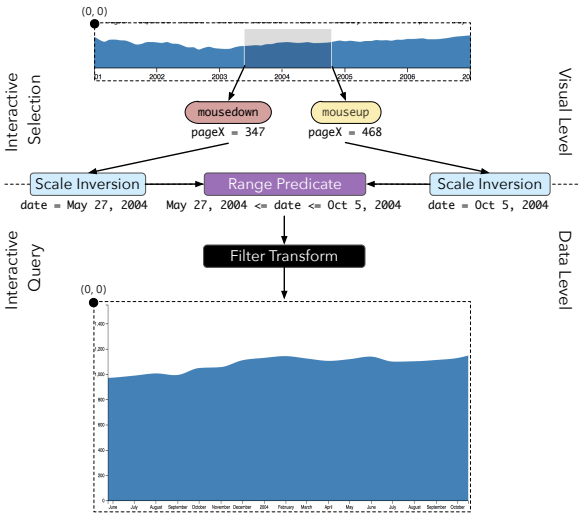

Figure 4. *Predicates* use signal values to define interactive selections of elements. Using *scale inversion*, predicates can be generalized to define interactive queries, and thus operate across different coordinate spaces in the same visualization: context (top) and focus (bottom).

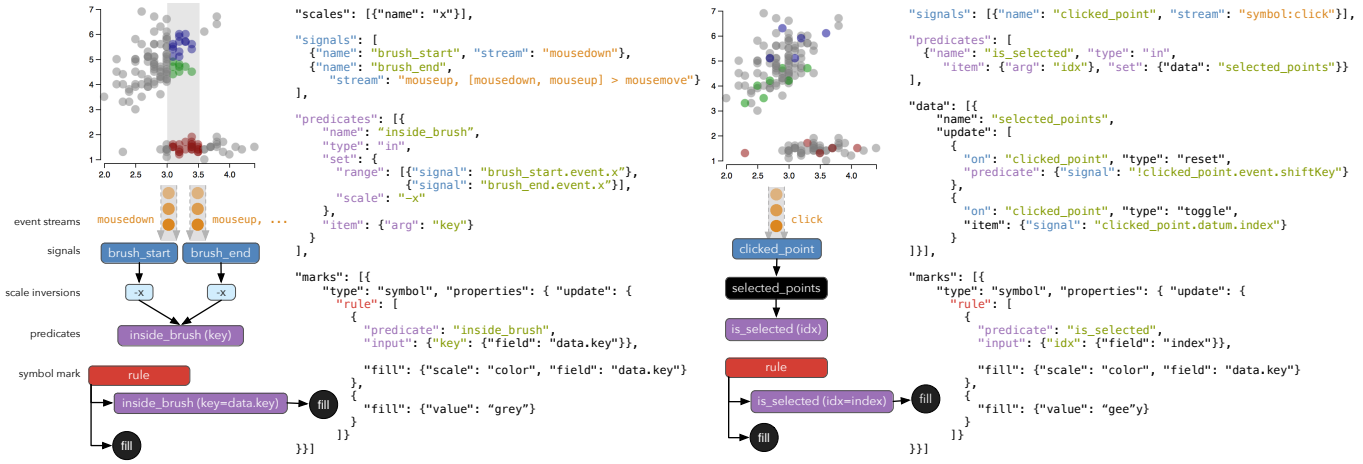

Figure 5. Expressing the selections in Figure 3 with our extensions to Vega: (left) brushing (only one dimension shown); (right) click/shift-click selection. Signal definitions and usage are in blue, event streams in orange, predicate definition and application in purple, rules in red, and names in green.

ple, sorting a co-occurrence matrix by frequency or applying a clustering algorithm to place similar rows and columns in close proximity. It is not feasible for a declarative language to natively support all possible functions, yet custom operations must still be expressible. Following the precedent of languages such as SQL, we provide *user-defined functions*. These functions can be declaratively invoked within the specification, akin to applying predicates, but must be registered and defined at runtime. User-defined functions ensure that the language remains concise and domain-specific, while ensuring extensibility to idiosyncratic operations.

Encapsulated Interactors

To allow reuse of custom interaction techniques, our model's primitives can be parameterized and encapsulated as named *interactors*. Interactors can subsequently be applied to containers or groups, or the top-level visualization, and function like mixins. An interactor's specification is merged into the host specification and, to prevent conflicts, its components are addressable only under its namespace.

When applied to the top-level visualization, only one instance of the interactor (and its primitives) is created. When applied to groups, however, the interactor is scoped to the group and thus multiple instances of the interactor may exist (one per group instance). In such cases, the signals and predicates of an interactor instance may be referenced using the corresponding group's key. However, it is often more useful to be able to automatically aggregate across instances. To facilitate this, when referencing an interactor's predicate, `latest`, `any`, or `all` flags use the most recently activated interactor, or perform logical `OR` or `AND` aggregates over all interactor instances, respectively.

Figure 6 illustrates how a brush interactor, extracted from Figure 5, can be applied to create a scatterplot matrix. Six instances of the brush interactor exist, one for each cell. The `latest` flag ensures that the `inside_brush` predicate is taken from the most recently used interactor, as determined by which cell the interaction occurs in. If we wished to allow multiple brushes, we would use the `all` flag instead.

IMPLEMENTATION

We implemented our model through reactive extensions to the Vega visualization grammar [36]. While our model can be instantiated under other visualization toolkits (e.g., D3 [6]), we chose Vega as its visualizations are purely declarative JSON specifications. As a result, reactive extensions to Vega allow us to assess the extent to which our model enables *declarative* interaction design. It is important to note that, by implementing our model, we seek to evaluate its feasability and expressivity (rather than the effectiveness of specific Vega syntax).

In Vega, the properties of graphical *marks* such as bars, lines, arcs, and text labels are bound to the attributes of backing data objects. Data sets can be specified inline or imported from a URL. The grammar includes data transformation operations, including common statistical summaries and visual layout algorithms such as force-directed layout, treemaps, and cartographic projections. Following the Grammar of Graphics [41], scales and guides (i.e., axes and legends) are provided as first-class objects. A JavaScript runtime parses JSON Vega specifications into web-based components rendered using HTML5 Canvas or SVG. Vega specifications can also be rendered server-side to produce static PNG or SVG files. Figure 1 illustrates how these abstractions are used to create a bar chart in Vega.

Vega uses a multi-stage pipeline for constructing visualizations, similar to Protovis [5]. A Vega JSON specification is parsed into a `View` object that manages data sets, data transformation pipelines, and visual encoding functions that determine the properties of graphical marks. In the build phase, an abstract *scenegraph* of marks is constructed with one node for each datum per mark; encoder functions then modify node properties. Finally, the abstract scenegraph is rendered, for example to an HTML5 Canvas.

We augment the parser to support our model's primitives and construct the necessary dataflow graph. Event listeners are registered to capture input events, and serve as source nodes in this graph. The `View` object is the sole sink. The remaining nodes in the graph are signals: either named signals from

the specification, or anonymous signals automatically constructed when a signal is used in the specification of another primitive, for example within a predicate or mark property.

Events are propagated using a push-based model [3]. When a source node receives an event, it pushes the event to all dependents. When a signal value changes, corresponding encoders are re-evaluated in a localized update to the visualization scenegraph. If a signal value triggers a change in a data set (for example, a signal feeds a predicate used in a filter transform), the corresponding transform pipeline and implicated scengraph branches are updated. To prevent wasteful computation and momentary inconsistencies — known as reactive *glitches* [8] — updates execute in topological order: a signal changes only when all of its parents are up-to-date.

Vega features an extensible data transformation pipeline; we reuse this feature to implement user-defined functions (UDFs). UDFs are named JavaScript functions that are registered with the Vega runtime, and can then be declaratively referenced in the specification, like built-in transformations.

We leave performance optimization to future work. However, all example visualizations render without visible lags using Google Chrome 34 on an Apple MacBook Pro with a 2.3 GHz Intel Core i7 processor and 8 GB of RAM.

EXAMPLE INTERACTIVE VISUALIZATIONS

To evaluate the expressivity of our language, we present a range of examples and demonstrate coverage over Yi et al.'s interaction taxonomy [42]. Yi et al. identify seven categories based on user intent: *select*, to mark items of interest; *explore*, to examine a different subset of data; *reconfigure*, to show a different arrangement of data; *encode*, to use a different visual encoding; *abstract/elaborate*, to show more or less detail; *filter*, to show something conditionally; and, *connect*, to show related items. It is important to note that these categories are not mutually exclusive, and an interaction technique can be classified under several categories. We choose example interactive visualizations to demonstrate that our model can express interactions across all seven categories and how, through composition of its primitives, supports the accretive design of richer interactions.

Select: Brushing and Click/Shift-Click

Figure 3 demonstrates two interaction techniques for selecting points. The first allows a user to brush over a region, and highlights points that lie within the brush extents. Signals are registered to capture the start and end positions of the brush, by default `mousedown` and `[mousedown, mouseup] > mousemove`, respectively. A range predicate uses scale inversion to test whether a point's data value falls within the specified horizontal and vertical data ranges. Figure 5 (left) illustrates how a one-dimensional version of this interaction can be expressed in Vega JSON.

The second selection technique allows the user to select individual points of interest. A signal over clicked points feeds data transforms that add or toggle points in a new "selected" data source. A predicate checks if the `shiftKey` is set and, if not, clears the data source before adding points. Another predicate is then used to highlight points that exist within the data source. Figure 5 (right) contains the Vega JSON for this interaction. Both selection techniques use a production rule to set the fill color of selected points.

Connect: Brushing & Linking

We can extend the previous example to create an encapsulated interactor for brushing and linking. We can then apply the interactor to add brushing to the scatterplot matrix shown in Figure 6. Each cell of the matrix is an instance of a group mark with its own coordinate space. The plotting symbol, and necessary spatial scale functions, are specified within this group. To brush within a cell of the matrix, we apply the brush interactor to the group's specification, and pass in the

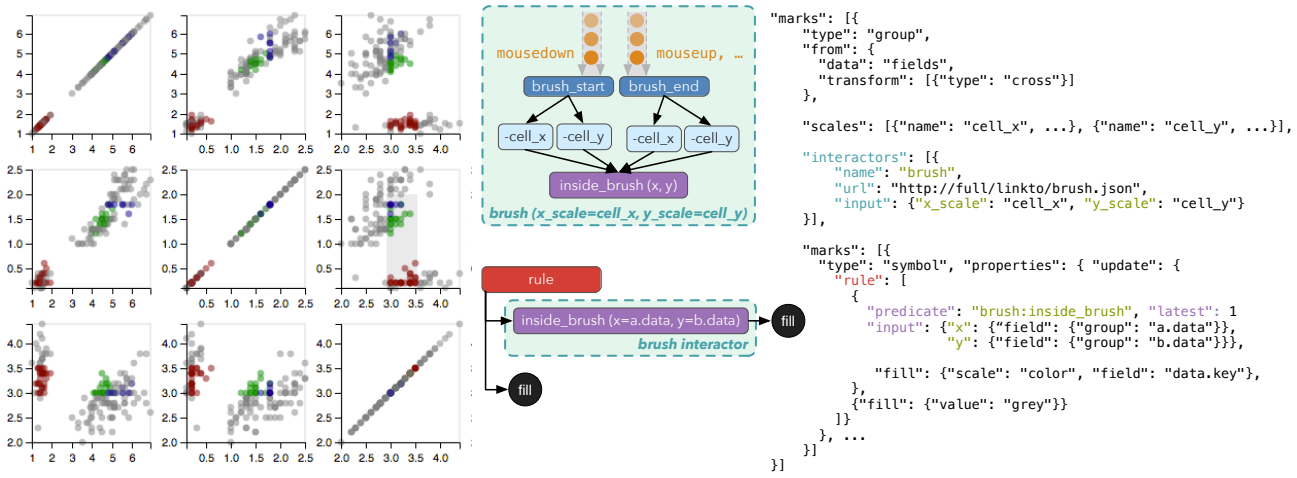

Figure 6. We can extract and parameterize the brushing interaction into an interactor, and apply it to the groups that define each cell of the scatterplot matrix. Primitives within the interactor can be referenced under its namespace.

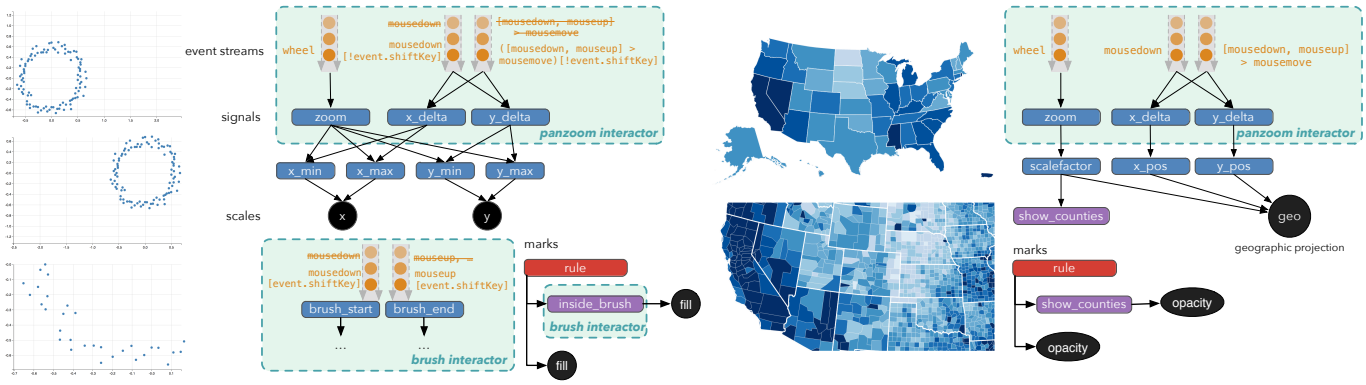

Figure 7. (left) Panning & zooming a scatterplot. Brushing can be added accretively by including the `brush` interactor, and rebinding event streams (indicated with strikethroughs). By extracting pan & zoom interactions to an interactor, we can repurpose it to perform semantic zooming on a geographic map (right). A map of the United States shows a choropleth of state-level unemployment. Once the map is zoomed past a threshold, states break up into counties, and show a choropleth of county-level unemployment.

appropriate scales. Signals and predicates defined within the interactor are addressable under its namespace.

Had the interactor predicates defined selections over pixel space, the production rule would only highlight points within the same cell as the brush (as each cell has a different coordinate space). However, as shown in Fig. 5 (left), the interactor predicates use scale inversion to lift the selection into the data domain. Thus, the production rule correctly performs the linking operation across scatter plots.

Abstract/Elaborate: Overview+Detail

With our brush interactor, we can also create the overview + detail visualization shown in Figure 4. In this case, brushing is restricted to the horizontal dimension. In our visualization, we override the `height` property of the visual brush added by the interactor, and ignore the vertical range predicates it populates. We use the horizontal range predicate with a filter transformation, to filter points for display in the detail plot. As a user draws a brush, signals update the horizontal range predicate, which in turn reactively filters points in the data source, updates scale functions and re-renders the detail view.

Explore and Encode: Panning & Zooming

Figure 7 (left) shows pan and zoom interactions for a scatter plot. By default, scale functions calculate their domain automatically from a data source. For this interaction, however, we must parameterize the domain using reactive signals. For panning, a `start` signal captures an initial `(x,y)` position, and subsequent `pan` signals calculate a delta. This delta is used to offset the scale domains. Similarly, a `zoom` signal applies a scale factor to the domains depending on the zoom direction. By default, these signals are mapped to `mousedown`, `[mousedown, mouseup] > mousemove` and `mousewheel`, respectively.

When adding a brushing interaction to this visualization, the brush signals may conflict with the signals for panning: on drag, both interactions might fire. One option to resolve this conflict is to begin brushing only when the `shiftKey` is depressed. If we try combining these interactions using D3 [6], which offers brushing and panning as part of its interactor typology, the process can be onerous. Additional callbacks must be registered that either instantiate or destroy a particular interaction depending on the `shiftKey` state.

In our model, the brush and pan signals can be rebound without modifying the interactor's definition. Instead, we provide alternate source event streams when instantiating the interactor. For example, `mousedown[event.shiftKey]` can drive the brush start signal, while the pan start signal can be triggered by `mousedown[!event.shiftKey]`. Similarly, the signals could be rebound to support touch interaction: one-finger drag to brush, two-finger drag to pan, and pinch to zoom.

We can also extract the pan & zoom definitions into their own interactor, and apply it to instead trigger semantic zooming [31], an *encoding* interaction technique shown in Figure 7 (right). At the top-level, the visualization shows a choropleth map of state-level unemployment. After crossing a specified zoom threshold, states subdivide to show a choropleth map of country-level unemployment. Here, the pan signals drive the geographic projection's translation and the zoom signals drive the projection's scale parameter. By default, both maps are drawn with states overlaying counties. A production rule uses a predicate to test whether the zoom signal is above a specified threshold; if it is, the state-level map is rendered transparently, displaying the county-level map underneath it.

Reconfigure: Index Chart

Figure 8 (left) shows an index chart: a line chart that interactively normalizes time series to show percentage change based on the current index point. To calculate the index point, we construct a signal over `mousemove` events and then drive the `x` coordinate through a scale inversion. As it is a quantitative scale, scale inversion results in a value from a continuous domain (i.e., any date/time from Jan 1 2000–Dec 31 2010). However, our dataset only contains stock prices for the start of every month, with the line interpolating between these points. To use scale inversion for an index point, we need to find the nearest data value. We build predicates that, for each time series, find a point that falls within a 2-week window on either side of the inverted point and use this as our index point. Using Vega's data transformations, we join the index point

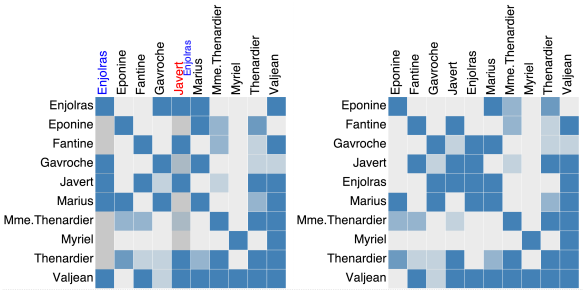

Figure 9. Re-arranging the columns of a co-occurrence matrix. The blue-headed column, Enjolras, is being dragged (left) and dropped over the red-headed column, Javert (right).

against the original data set and normalize the data values. Scale functions are defined in terms of the normalized data.

As a user moves their mouse across the index chart, the signal and scale inversion automatically update, data transforms find a new index point and normalize the data, scale functions update, and the visualization is re-rendered. This example demonstrates how interaction can be used in a feedback loop to provide data values that drive the visualization.

Reconfigure: Reordering Columns of a Matrix

Figure 9 shows a co-occurrence matrix of Les Misérables characters. To reorder the columns of the matrix, we first construct a data source that models the sort order of characters and initialize it to an alphabetical ordering. A signal on `.col_label:mousedown` captures the source column to be reordered, while a signal on `[.col_label:mousedown, mouseup] > mousemove` updates the target column location. On `mouseup`, the data source is updated using a UDF to remove the source column, and reinsert it at the target column's index.

Filter: Control Widgets

Figure 8 (right) shows the Job Voyager [19] visualization with control widgets to filter the visualized data. A textbox allows users to enter search terms to filter job titles, while the radio buttons allow users to filter by gender. We bind signals to the value of these control widgets, and then construct predicates attached to filter data transformations. For the textbox signal, a match predicate tests search terms against job titles, while an equality predicate filters by gender based on the radio button signal. This example illustrates how external query widgets can easily be bound into our reactive interaction model.

DISCUSSION: COGNITIVE DIMENSIONS OF NOTATION

The example interactive visualizations in the previous section demonstrate our model's expressiveness. Here, we seek to evaluate our model from a designer's standpoint. We use the Cognitive Dimensions of Notation [4], which provides a set of heuristics for evaluating the efficacy of notational systems such as programming languages and interfaces. Of the 14 dimensions, we evaluate our model against a relevant subset, and primarily compare it against common practices: declarative specification of visual encoding using D3 [6] and imperative event handling callbacks for interaction.

Abstractions (types and availability of abstraction mechanisms) and *Viscosity* (resistance to change). Streams and signals abstract the low-level events that trigger interactions and decouple them from downstream logic. This approach can facilitate rapid iteration: the result of an interaction can be designed (for example, highlighting points), and then a variety of different event triggers can be prototyped by simply rebinding the appropriate signals. As our examples demonstrate, binding signals reduces the burden for resolving conflicting interactions or retargeting to different platforms. By comparison, iterating with event callbacks can be more difficult. For example, a specific sequence of events may require a specific ordering of callbacks, and coordinating the visualization state across these various sequences falls to the designer.

Premature Commitment (constraints on the order of doing things). Abstracting streams and signals does impose a premature commitment. Users must create them before they are able to use any lower-level events to trigger state changes. This requirement could be relaxed: users could use low-level streams and signals inline, for example in a predicate definition. However, we believe inline reference to low-level events streams is a poor design pattern, as it makes an interaction technique dependent on a specific set of events, a common problem with existing interactor typologies. This pattern re-

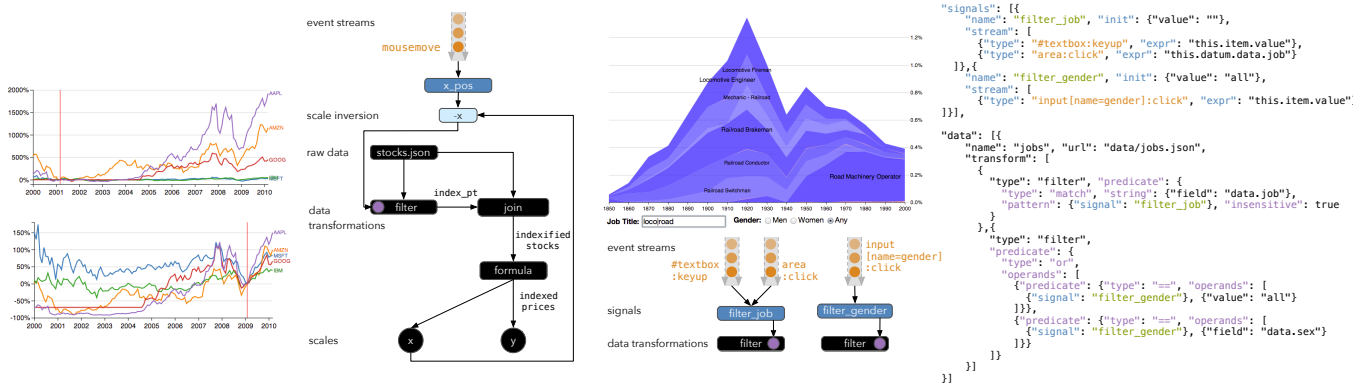

Figure 8. (left) An index chart that shows the percentage changes for a collection of time-series. The index point (red vertical line) is determined by the x position of a `mousemove` signal. (right) The job voyager can be filtered using signals bound to the state of control widgets. A textbox allows for pattern-based filtering of job titles while radio buttons allow gender-based filtering.

duces reusability, making it more difficult to resolve conflicting interactions (e.g., brushing vs. panning) or retarget interactions across input modalities.

Hidden Dependencies (important links between entities are not visible). Abstracting streams and signals also introduces hidden dependencies, as they obscure which input events are triggering a particular visual state change. However, we believe that the viscosity advantages outweigh the complexities of added hidden dependencies, which can be alleviated by naming the abstractions appropriately. Furthermore, as our code examples illustrate, under our model all the factors that directly affect a particular state are captured within a single specification. For example, a signal definition specifies all events or signals that may affect its value; similarly, a visual property may use a rule which enumerates all the values it may take. With D3, the visual specification may not completely define all visualization states. Instead, the user must trace a flow through event callbacks, a process further exacerbated by unpredictable execution order. The user is forced to coordinate interleaved callbacks, introducing *hard mental operations* and *error-proneness*.

Consistency (similar semantics are expressed in similar syntactic forms). Our interaction model is best suited for systems that declaratively specify visual encodings, and would feel foreign in imperative systems. However, given the widespread adoption of D3, and Vega's increasing integration in systems [14, 22, 34] we believe this is not a significant liability. By contrast, registering event callbacks on D3 visualizations breaks consistency: visual design is declarative while interaction design is imperative. It requires users to think in terms of different notational systems, and exposes underlying implementation and execution concerns.

Visibility (ability to view components easily). One of the primarily advantages of using D3, and registering event callbacks, is being able to debug code directly within a web browser [6]: the generated visualization can be inspected, while the JavaScript console can be used to interactively debug event callbacks. However, this is more difficult with Vega: its runtime environment, which parses and renders a specification, introduces its own stack of runtime abstractions. It is important to note that this is a limitation of the current implementation of our model, rather than an inherent limitation in the model itself. For example, our model might also be used to implement extensions to D3, thus gaining the advantages of declarative interaction design without losing debugging capabilities.

In summary, our model introduces some hidden dependencies and decreases visibility. However, we believe these are outweighed by the increase in specification consistency of visual encoding and interaction, and the decrease in viscosity by supporting abstraction mechanisms.

CONCLUSION

Our model contributes a substantive step towards enabling declarative interaction design for data visualization. An important next step is to assess the language's accessibility through user evaluations. Are new users able to learn this model? Can experts accustomed to callback-driven programming quickly transition to a reactive model? Our work here primarily focuses on the design of model primitives, and we only implemented cursory optimizations of our reactive dataflow graph. While existing performance is adequate for many common visualization scenarios, more extensive performance optimization is possible. For example, insights from the streaming data literature [1] might be applied to optimize interactive queries.

Vega's declarative format has facilitated integration into a variety of systems, including graphical design tools [34], statistical packages [14], and computational environments [22]. We anticipate that our model's declarative approach, and the ability to package reusable techniques as interactors, will facilitate similar integration opportunities. We intend to contribute our extensions back into the public Vega project. By reducing the burden for programmatic generation of interactions, we also hope to spur study into alternate, non-textual, ways of specifying interactions; for example, interaction design through demonstration or by direct manipulation.

Learning and adapting examples is an important part of the design process [20, 35]. Designers can use our notion of encapsulated, standalone interactors, to share, reuse, and learn from the design of others. Corpora of interaction designs could be created — similar to `bl.ocks.org` with D3 [6] visualizations — allowing users to browse through designs for inspiration, or adapt them for their own visualizations.

ACKNOWLEDGMENTS

This work was supported by an SAP Stanford Graduate Fellowship, the Intel ISTC for Big Data, and DARPA XDATA.

REFERENCES

1. Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., and Widom, J. Stream: The stanford data stream management system. *Book chapter* (2004).
2. Badros, G. J., Borning, A., and Stuckey, P. J. The cassowary linear arithmetic constraint solving algorithm. *ACM Trans. on Computer-Human Interaction (TOCHI) 8*, 4 (2001), 267–306.
3. Bainomugisha, E., Carreton, A. L., Cutsem, T. v., Mostinckx, S., and Meuter, W. d. A survey on reactive programming. *ACM Computing Surveys (CSUR) 45*, 4 (2013), 52.
4. Blackwell, A. F., Britton, C., Cox, A., Green, T. R., Gurr, C., Kadoda, G., Kutar, M., Loomes, M., Nehaniv, C. L., Petre, M., et al. Cognitive dimensions of notations: Design tools for cognitive technology. In *Cognitive Technology: Instruments of Mind*. Springer, 2001, 325–341.
5. Bostock, M., and Heer, J. Protovis: A graphical toolkit for visualization. *IEEE Trans. Visualization & Comp. Graphics 15*, 6 (2009), 1121–1128.
6. Bostock, M., Ogievetsky, V., and Heer, J. D3: Data-Driven Documents. *IEEE Trans. Visualization & Comp. Graphics 17*, 12 (2011), 2301–2309.

7. Cooper, G. H. *Integrating dataflow evaluation into a practical higher-order call-by-value language*. PhD thesis, Brown University, 2008.
8. Cooper, G. H., and Krishnamurthi, S. Embedding dynamic dataflow in a call-by-value language. In *Programming Languages and Systems*. Springer, 2006, 294–308.
9. Cottam, J., and Lumsdaine, A. Stencil: a conceptual model for representation and interaction. In *Information Visualisation*, IEEE (2008), 51–56.
10. Czaplicki, E., and Chong, S. Asynchronous functional reactive programming for guis. In *Proc. ACM SIGPLAN*, ACM (2013), 411–422.
11. Derthick, M., Kolojejchick, J., and Roth, S. F. An interactive visual query environment for exploring data. In *Proc. ACM UIST*, ACM (1997), 189–198.
12. Edwards, J. Coherent reaction. In *Proc. ACM SIGPLAN*, ACM (2009), 925–932.
13. Elliott, C., and Hudak, P. Functional reactive animation. In *ACM SIGPLAN Notices*, vol. 32, ACM (1997), 263–273.
14. ggvis: Interactive grammar of graphics for R. `http://ggvis.rstudio.com/`, April 2014.
15. Heer, J., and Agrawala, M. Software design patterns for information visualization. *IEEE Trans. Visualization & Comp. Graphics 12*, 5 (2006), 853–860.
16. Heer, J., Agrawala, M., and Willett, W. Generalized selection via interactive query relaxation. In *Proc. ACM CHI*, ACM (2008), 959–968.
17. Heer, J., and Bostock, M. Declarative language design for interactive visualization. *IEEE Trans. Visualization & Comp. Graphics 16*, 6 (2010), 1149–1156.
18. Heer, J., Card, S. K., and Landay, J. A. Prefuse: a toolkit for interactive information visualization. In *Proc. ACM CHI*, ACM (2005), 421–430.
19. Heer, J., Viégas, F. B., and Wattenberg, M. Voyagers and voyeurs: supporting asynchronous collaborative information visualization. In *Proc. ACM CHI*, ACM (2007), 1029–1038.
20. Herring, S. R., Chang, C.-C., Krantzler, J., and Bailey, B. P. Getting inspired!: understanding how and why examples are used in creative design practice. In *Proc. ACM CHI*, ACM (2009), 87–96.
21. Hudson, S. E., and Smith, I. Ultra-lightweight constraints. In *Proceedings of the 9th annual ACM symposium on User interface software and technology*, ACM (1996), 147 –155.
22. The IPython Notebook. `http://ipython.org/notebook.html`, April 2014.
23. Kin, K., Hartmann, B., DeRose, T., and Agrawala, M. Proton++: a customizable declarative multitouch framework. In *Proc. ACM UIST*, ACM (2012), 477–486.
24. Lee, E. A., and Messerschmitt, D. G. Synchronous data flow. *IEEE Proc. 75*, 9 (1987), 1235–1245.
25. Liu, Z., and Stasko, J. T. Mental models, visual reasoning and interaction in information visualization: A top-down perspective. *IEEE Trans. Visualization & Comp. Graphics 16*, 6 (2010), 999–1008.
26. Meyerovich, L. A., Guha, A., Baskin, J., Cooper, G. H., Greenberg, M., Bromfield, A., and Krishnamurthi, S. Flapjax: a programming language for ajax applications. In *ACM SIGPLAN Notices*, vol. 44, ACM (2009), 1–20.
27. Myers, B. A. A new model for handling input. *ACM Trans. on Information Systems (TOIS) 8*, 3 (1990), 289–320.
28. Myers, B. A. Separating application code from toolkits: eliminating the spaghetti of call-backs. In *Proc. ACM UIST*, ACM (1991), 211–220.
29. Oney, S., Myers, B., and Brandt, J. Constraintjs: programming interactive behaviors for the web by integrating constraints and states. In *Proc. ACM UIST*, ACM (2012), 229–238.
30. Ortiz, S., and Cid, V. P. Use cases of impure, an information interface, 2010.
31. Perlin, K., and Fox, D. Pad: an alternative approach to the computer interface. In *Proc. Comp. Graphics & Interactive Techniques*, ACM (1993), 57–64.
32. Pike, W. A., Stasko, J., Chang, R., and O'Connell, T. A. The science of interaction. *Information Visualization 8*, 4 (2009), 263–274.
33. Quadrigram. `http://www.quadrigram.com/`, April 2014.
34. Satyanarayan, A., and Heer, J. Lyra: An interactive visualization design environment. *Computer Graphics Forum (Proc. EuroVis)* (2014).
35. Schön, D. A. *The reflective practitioner: How professionals think in action*, vol. 5126. Basic books, 1983.
36. Vega: A Visualization Grammar. `http://trifacta.github.io/vega`, April 2014.
37. Victor, B. Drawing Dynamic Visualizations. `http://vimeo.com/66085662`, February 2013.
38. Wan, Z., Taha, W., and Hudak, P. Event-driven FRP. In *Practical Aspects of Declarative Languages*. Springer, 2002, 155–172.
39. Weaver, C. Building highly-coordinated visualizations in Improvise. In *Proc. IEEE Information Visualization* (2004), 159–166.
40. Wickham, H. *ggplot2: Elegant Graphics for Data Analysis*. Springer, 2009.
41. Wilkinson, L. *The Grammar of Graphics*. Springer, 2005.
42. Yi, J. S., ah Kang, Y., Stasko, J. T., and Jacko, J. A. Toward a deeper understanding of the role of interaction in information visualization. *Visualization and Computer Graphics, IEEE Transactions on 13*, 6 (2007), 1224–1231.