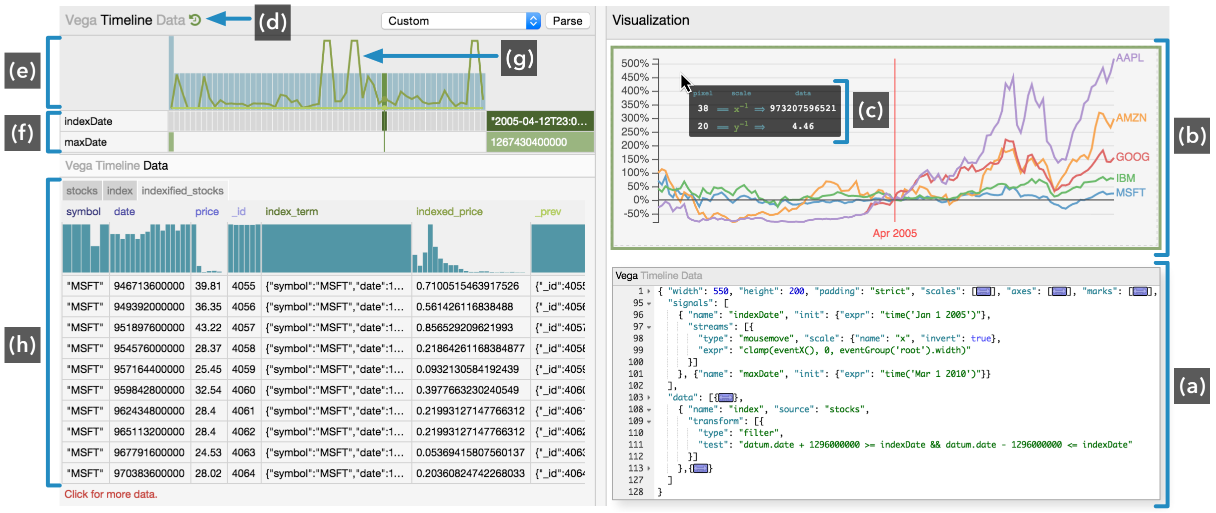
Eurographics Conference on Visualization (EuroVis) 2016
K.-L. Ma, G. Santucci, and J. van Wijk
(Guest Editors)

Volume 35 (2016), Number 3

Visual Debugging Techniques for Reactive Data Visualization

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

[1] University of Washington [2] Stanford University



symbol	date	price	_id	index_term	indexed_price	_prev
"MSFT"	946713600000	39.81	4055	{"symbol":"MSFT","date":1...	0.7100515463917526	{"_id":4055
"MSFT"	949392000000	36.35	4056	{"symbol":"MSFT","date":1...	0.561426116838488	{"_id":4056
"MSFT"	951897600000	43.22	4057	{"symbol":"MSFT","date":1...	0.856529209621993	{"_id":4057
"MSFT"	954576000000	28.37	4058	{"symbol":"MSFT","date":1...	0.21864261168384877	{"_id":4058
"MSFT"	957164400000	25.45	4059	{"symbol":"MSFT","date":1...	0.0932130584192439	{"_id":4059
"MSFT"	959842800000	32.54	4060	{"symbol":"MSFT","date":1...	0.3977663230240549	{"_id":4060
"MSFT"	962434800000	28.4	4061	{"symbol":"MSFT","date":1...	0.21993127147766312	{"_id":4061
"MSFT"	965113200000	28.4	4062	{"symbol":"MSFT","date":1...	0.21993127147766312	{"_id":4062
"MSFT"	967791600000	24.53	4063	{"symbol":"MSFT","date":1...	0.05369415807560137	{"_id":4063
"MSFT"	970383600000	28.02	4064	{"symbol":"MSFT","date":1...	0.20360824742268033	{"_id":4064

```
{ "width": 550, "height": 200, "padding": "strict", "scales": [], "axes": [], "marks": [],
  "signals": [
    { "name": "indexDate", "init": {"expr": "time('Jan 1 2005')"},
      "streams": [{
        "type": "mousemove", "scale": {"name": "x", "invert": true},
        "expr": "clamp(eventX(), 0, eventGroup('root').width)"
      }]
    }, {"name": "maxDate", "init": {"expr": "time('Mar 1 2010')"}}
  ],
  "data": [{},
    { "name": "index", "source": "stocks",
      "transform": [{
        "type": "filter",
        "test": "datum.date + 1296000000 >= indexDate && datum.date - 1296000000 <= indexDate"
      }]
    },{}
  ]
}
```

Figure 1: *Visual debugging techniques enable inspection of program state and interaction logic for a reactive data visualization in Vega. Designers author (a) a declarative specification to produce (b) an interactive visualization. (c) Tooltips on the visualization provide introspection into visual encodings while viewing a past state via (d) replay. Recorded interactions are shown in (e) an overview and (f) a timeline. (g) A time series shows the variability of data attributes in (h) the backing datasets.*

Abstract

Interaction is critical to effective visualization, but can be difficult to author and debug due to dependencies among input events, program state, and visual output. Recent advances leverage reactive semantics to support declarative design and avoid the "spaghetti code" of imperative event handlers. While reactive programming improves many aspects of development, textual specifications still fail to convey the complex runtime dynamics. In response, we contribute a set of visual debugging techniques to reveal the runtime behavior of reactive visualizations. A timeline view records input events and dynamic variable updates, allowing designers to replay and inspect the propagation of values step-by-step. On-demand annotations overlay the output visualization to expose relevant state and scale mappings in-situ. Dynamic tables visualize how backing datasets change over time. To evaluate the effectiveness of these techniques, we study how first-time Vega users debug interactions in faulty, unfamiliar specifications; with no prior knowledge, participants were able to accurately trace errors through the specification.

Categories and Subject Descriptors (according to ACM CCS): H.5.2 [Information Interfaces]: User Interfaces—GUI

1. Introduction

Interaction techniques such as filtering, brushing, and dynamic queries facilitate data exploration and understanding [HS12, PSCO09]. However, implementing such interactions has traditionally required event callbacks, which necessitate manually tracking interleaved state changes [Mye91].

In response, recent work [SWH14, SRHH15] leverages event-driven functional reactive programming [WTH02] to provide declarative primitives for interaction design. This approach models input events as data streams, which in turn drive dynamic variables called *signals*. Signals parameterize the remainder of the visualization, endowing transforms, scales, and marks with reactivity. When new input events fire, corresponding signals are automatically re-evaluated. Updates propagate to visual encodings and the visualization is re-rendered. By deferring low-level control flow to the system, declarative visualization languages can enable rapid iteration of encoding and interaction design.

However, when interactions produce erroneous results, existing debugging techniques such as breakpoints or stack traces are no longer effective since users are unfamiliar with the underlying control flow. Therefore, new debugging techniques are needed to understand relevant state changes and assess breakdowns. The welldefined semantics of declarative visual encodings provide new opportunities for enhanced debugging support, as tools can surface traces from pixels, through scale transforms, to source data (and vice versa). Regardless of programming style, interactions can be inherently difficult to author and debug. Developers must understand complex dependencies among input events, program state, and visual output. Textual specifications alone are inadequate for tracking relationships through time-varying behaviors. To debug faulty interactions, developers must inspect the **state** of events and program variables during interaction, and track **changes** over time.

In this paper, we describe formative interviews with visualization developers to assess their debugging needs. We then contribute a set of visual debugging techniques for reactive data visualizations, motivated by three design goals to enable users to **probe the state**, **visualize relationships**, and **inspect transitions**.

Consider debugging an index chart of stock prices that interactively renormalizes the data based on the mouse position (Fig. 1b, 2). A user first writes a *specification* (Fig. 1a) of encoding rules and interactions. During interaction, the user notices that at certain time points, all the time series flatline (Fig. 4b) due to a specification error. The user must now assess the dependencies between interaction, program state, and visual output. She could start by recording interactions in the *timeline* (Fig. 1f), and *replaying* (Fig. 1d) to observe how events propagate. The *overview* (Fig. 1e) summarizes activity, allowing for quick identification of interaction patterns. *In-situ annotations* (Fig. 1c) expose the faulty position encoding by showing the data values and encodings corresponding to the selected pixel. The user can then inspect the backing dataset via *dynamic tables* (Fig. 1h). Guided by the *attribute variability* (Fig. 1g), she observes that some data attributes have been zeroed out, which she selects to link back to the specification to fix the error.

We instantiate these techniques in the context of Vega [SWH14], a declarative visualization grammar that supports reactive interaction design. In an initial evaluation, we study how 12 first-time Vega users debug faulty interactions in unfamiliar specifications. Despite their lack of expertise with Vega, we find that the participants can accurately trace errors to problematic lines in the specifications by employing our visual debugging techniques.

2. Related Work

Our visual debugging techniques leverage event-driven functional reactive programming abstractions, and are informed by prior work on timeline- and replay-based interactive debuggers, and visual representations of program state and behavior.

2.1. Functional Reactive Programming

Event-Driven Functional Reactive Programming (E-FRP) [WTH02], one of many FRP variants [BCC*13], is an increasingly popular paradigm for authoring interactive behaviors. E-FRP models low-level input events as continuous streams of data, which can be composed into dynamic variables called *signals*. When a new event fires, the E-FRP runtime propagates it to the corresponding streams, and dependent signals are updated in two phases. In the first phase, signals are reevaluated using their dependencies' prior values; these dependencies are then reevaluated in the second phase [WTH02]. E-FRP has been shown to be suitably expressive for interactive web applications [MGB*09, CC13] and visualizations [CL08, KL15, SWH14]. In this section, we focus on the former and defer the latter to the subsequent section.

Although E-FRP is sufficiently expressive for web applications, debugging support remains weak. Many existing debugging techniques — such as breakpoints and stack traces — no longer apply, as users *declaratively* specify interactions. The E-FRP runtime is entirely responsible for the program execution, the particulars of which will be unfamiliar to end-users. The Elm language [CC13] has begun to develop an interactive debugger, inspired by Bret Victor [Vic12]. The Elm debugger allows recording and replaying program states, but developers must manually annotate their code with `watch` and `trace` statements. Tracked states are then simply printed out in a list. In contrast, our timeline view automatically tracks *all* user-defined signals. Along with the overview, the timeline provides users with a visual representation of event and state propagation, which facilitates identifying faulty behavior.

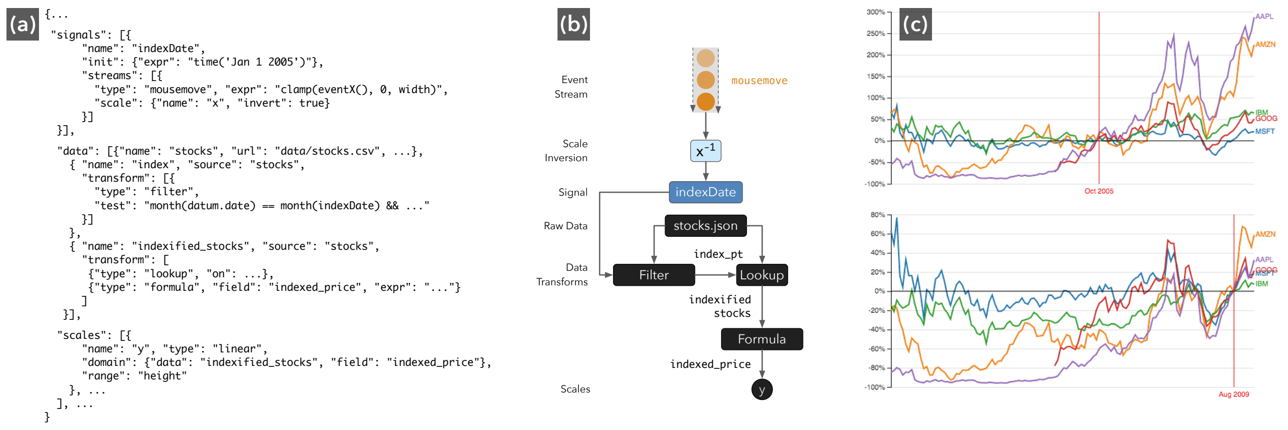
2.2. Timeline- and Replay-based Debuggers

The technique of recording and replaying program states can also be found in the FireCrystal [OM09] and Timelapse [BBKE13] systems. Both systems target interactive behaviors on web pages, but require significant supporting infrastructure. For example, to capture user interaction, FireCrystal must register a watcher on the Document Object Model (DOM) *and* a number of event listeners; it then leverages the Firefox browser's debugging API to identify which line of code is executed. The authors report that this operation is CPU-intensive and can affect interactive performance during recording [OM09]. For Timelapse, Burg et al. developed Dolos: extensions to the WebKit browser engine to record and replay interactions. By operating at this low-level, Dolos overcomes the overhead of watchers and event listeners, and integrates directly with the browser's existing JavaScript inspector [BBKE13].

By using E-FRP semantics, such complex infrastructures are not necessary to support our visual debugging techniques. In particular (as detailed in §3), we need only track and visualize signal values over time. This simplification is possible because signals express the bulk of an interaction technique, abstracting away the particular input events that trigger interactive behavior. Simple static analysis of the specification can then identify where signals are used.

2.3. Visual Representations of Program Behavior

Representing program state visually has been found to be a powerful pedagogical aid. Online Python Tutor [Guo13] provides visualizations of allocated Python objects, pointer references, global variables, and stack frames. Teachers have incorporated this system into course materials and reported that the diagrams mirrored ones they produce in class to help students build mental models. This finding was corroborated by students accessing the site as a supplementary learning tool, who shared positive anecdotes about its utility on online discussion forums. Whyline [KM04] and Theseus [LBM14] investigate the effects of introducing visualizations within integrated development environments. Whyline's extensions reduced debugging time by a factor of 8, while developers adopted



```
{...
 "signals": [{
     "name": "indexDate",
     "init": {"expr": "time('Jan 1 2005')"},
     "streams": [{
       "type": "mousemove", "expr": "clamp(eventX(), 0, width)",
       "scale": {"name": "x", "invert": true}
     }]
  }],
  "data": [{"name": "stocks", "url": "data/stocks.csv", ...},
    { "name": "index", "source": "stocks",
      "transform": [{
        "type": "filter",
        "test": "month(datum.date) == month(indexDate) && ..."
      }]
    },
    { "name": "indexified_stocks", "source": "stocks",
      "transform": [
       {"type": "lookup", "on": ...},
       {"type": "formula", "field": "indexed_price", "expr": "..."}
      ]
   }],
  "scales": [{
      "name": "y", "type": "linear",
      "domain": {"data": "indexified_stocks", "field": "indexed_price"},
      "range": "height"
    }, ...
  ], ...
}
```

Figure 2: *(a) An excerpt of the Vega* JSON *specification and (b) a corresponding design schematic for (c) an interactive index chart. Event streams capture* `mousemove` *events that are passed through an inverted scale transform and stored in a signal. The signal parameterizes data transforms to select an index point and normalize stock price time-series data.*

entirely new problem-solving strategies by leveraging Theseus' visualizations. These results motivate our study of *visual* techniques for debugging reactive visualizations.

System profilers also make use of visualizations. For example, Flame Graphs [Gre15] are widely used to understand the CPU and memory performance of applications, and similar visualizations have been added to profile JavaScript performance within the Chrome web browser [Goo15b]. Akin to GNU ggprof [GKM82], Chrome also provides graph visualizations to profile JavaScript's memory usage [Goo15a]. More recently, Perfopticon [MHHH15] visualizes the query plan and execution behavior of distributed database systems. Algorithm visualizations map low-level algorithmic behavior to visual properties, with animations showing changes over time [DFS02]. All of these visualizations have been purpose-built to expose low-level execution details, enabling expert users to identify performance bottlenecks. During our formative studies, experienced visualization designers reported that displaying the execution pipeline that produces the resultant visualization would only be tangentially useful when debugging faulty behavior. The user defers execution to the system, rendering low-level visualization techniques ineffective since users lack familiarity with the internal structure. Thus, our techniques maintain the abstraction level of the specification language used by users.

3. Background: The Vega Visualization Grammar

Our visual debugging techniques were developed in the context of Vega, a declarative visualization grammar. In this section, we briefly describe the aspects of Vega relevant to this paper. For concrete interactive examples, we refer interested readers to the online Vega editor: `http://vega.github.io/vega-editor`.

Closely following the model of Protovis [BH09] and D3.js [BOH11], Vega visualizations comprise graphical primitives called *marks*, such as *bars*, plotting *symbols*, and *lines*, whose properties are determined by the attributes of backing datasets. Integrated *data transformation* pipelines provide operations including statistical summarization and spatial layout (e.g., treemaps and cartographic projections). *Scales* map data attributes to visual variables, and are visualized by *guides* (i.e., axes and legends). Vega visualizations are expressed using JSON. A JavaScript runtime parses input specifications and produces resulting visualizations [SRHH15].

To support interaction design, Vega uses Event-Driven Functional Reactive Programming (E-FRP) [SWH14]. Input events are modeled as streams of data, and an event selector syntax facilitates stream composition. For example, `[mousedown, mouseup] > mousemove` specifies a stream of `mousemove` events that occur between `mousedown` and `mouseup` (otherwise known as *drag*). Event streams serve as a first-class data source. *Signals* are in turn defined as reactive expressions over stream values. For instance, a signal might extract the *x* and *y* coordinates from the most recent mouse input event. Signal values defined in pixel space can be passed through inverse scale transforms to map back to the data domain. Scale inversions allow interactive behaviors to generalize across distinct coordinate spaces (e.g., small multiples) or coordinate multiple visualizations (e.g., brushing and linking).

Signals can parameterize the remainder of the Vega specification, thereby endowing data transformations and visual encodings with reactive semantics (Fig. 2). Reactive updates (referred to as *pulses*) occur in two steps [WTH02]. When an event occurs, dependent signals are re-evaluated in their specification order. This step allows signal expressions to access the previous values of dependencies listed later in the specification; these dependencies are subsequently updated on the same pulse. Once the signals have updated, dependent data transformations and visual encodings are recomputed in topological order of the underlying dependency graph.

Signals are critical for enabling the development of our visual debugging techniques. Signals decouple low-level input events from interaction logic. For example, the same set of named signals can be driven by mouse and touch events. Moreover, signals express the bulk of the interaction logic and participate in visual encoding either as direct parameters or by parameterizing simple if-then-else encoding rules. As a result, signals provide a meaningful entry-point into an interaction specification. In contrast to imperative event handlers, complex static analysis is not required to identify and surface the relevant program state.

4. Formative Interviews & Design Goals

To better understand the debugging needs for reactive data visualization, we conducted formative interviews with Vega developers regarding their development processes. At the time of the study, Vega's reactive extensions had not yet been officially released, so participants were primarily familiar with static visualizations.

Prior to this work, there was no infrastructure for debugging visualizations in Vega. Users could only rely on the JavaScript console to traverse the underlying system internals. However, this method was not discoverable or intuitive for novice users. Accessing and navigating the system internals requires existing knowledge of how to locate relevant information, which is often deeply nested in the internal structure. This structure contains extraneous details that complicate identification of relevant information. The structural disconnect between signals, data, and encodings makes it hard to track changes between components, thus making it impractical for complex tasks. An example of the debugging process in this environment is available in the supplementary material.

Participants. We recruited 8 software professionals (all male), all with experience creating static Vega visualizations, and none affiliated with the University of Washington. Participants were selected based on their participation in the Vega community. Each interview lasted about 30 minutes; participants did not receive compensation.

Protocol. The semi-structured interviews examined each participant's development process as related to Vega. Participants were shown sample visualizations of Vega's dataflow graph and asked to reflect on the utility of such techniques with respect to their debugging needs; one participant was unable to access and view the sample visualizations during the interview. The full script is included in the supplementary material and includes the following questions:

- What was the last (or most troublesome) error you encountered when generating a Vega specification?
- In what ways do you think the debugging process could have been facilitated?
- Do you think that having the dataflow graph visualized would be useful for the development process?

Data Collection. The interviews took place over Skype and Google Hangouts. The example visualizations were shared using Google Docs. We captured audio recordings for later review and transcribed notes during the interview.

Results. Errors in encoding are often visually salient (e.g., points are filled with the wrong color), but tracing the error through the specification can be difficult — is the result due to an incorrect scale definition, an error in data transformations, or a problem in the input data itself? With Vega's declarative model, users lack visibility into the state of these components. One participant noted that "*when you mess up that* JSON *you get an error from deep in JavaScript land*," while another participant described difficult debugging scenarios where "*[the resultant visualization is] just blank and you don't know why*."

Participants noted that visualizing the internal dataflow graph could be beneficial for Vega *system* developers, but provides too much internal information tangential to their *user-level* debugging tasks. In particular, one participant noted that "*the [dataflow] graph presumes insight into how Vega's internals operate*." Inspecting the state via the JavaScript console or viewing Vega's dataflow graph presents users with a mixture of state information, only a small fraction of which is relevant to the debugging task at hand. The extraneous system details complicate identification of relevant information, suggesting that it would be beneficial to strip internal system information from the user's view.

Participants explained that their needs centered on the relationships between data and encodings expressed within their specifications. One participant explained that Vega "*need[s] a way to examine internal variables... [and] to see the internals of the step-by-step process*." Many participants additionally expressed the need to understand "*the structure of the data that Vega is actually using*" because data transformations may restructure the data or introduce new attributes. One participant noted that "*the easiest path to solve [the specification error] was to just break into the [JavaScript] debugger and see what state the data was in at various stages*."

Interactions further complicate the debugging process. Signals parameterize data transformations and encodings, introducing additional dependencies. While signals usefully abstract low-level input events, some users find that this abstraction complicates reasoning about event propagation. As one participant stated, "*debugging reactivity is like a true true nightmare*." Our interviews and observations regarding interaction inform three debugging design goals.

Probe the state: At a given moment, the visualization is determined by signal values, data transformations, and encoding rules. Users must be able to inspect the state of each of these components.

Visualize relationships: The state of one component often affects the state of others — for example, signals can parameterize encoding rules, or data transformations may affect scale domains. Users must be able to identify dependencies between components.

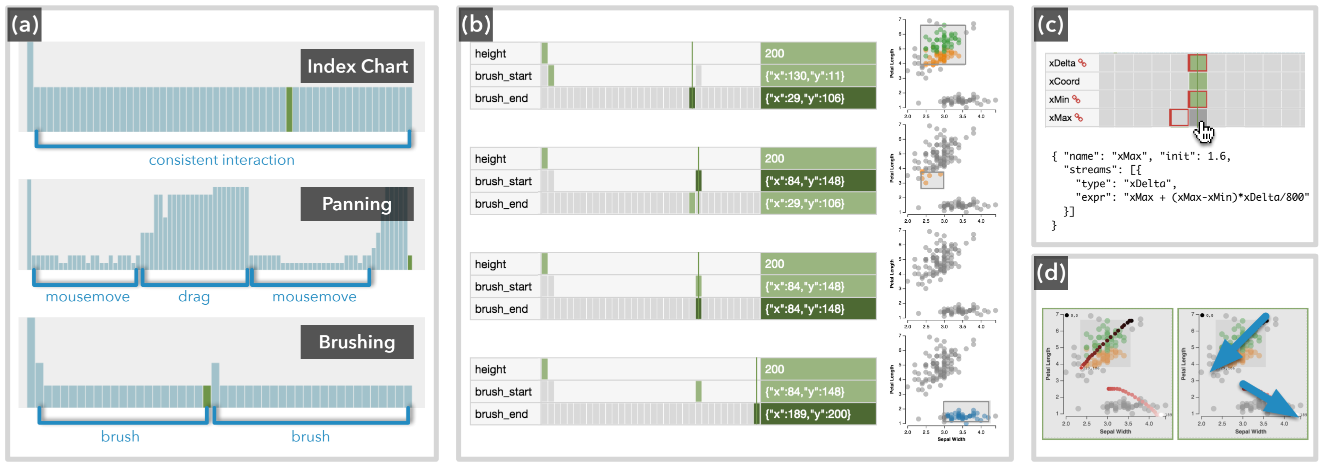
Inspect state transitions: Input events trigger transitions from state to state, and debugging faulty interactions requires understanding the causes and consequences of these transitions. To identify the source of an error, users must be able to inspect how values propagate through the specification.

5. Visual Debugging Techniques

We now present the design of our visual debugging techniques for reactive data visualization. In the formative studies, one participant observed that "*There are two possible errors. One is like a runtime error... The other is you actually have a well-formed execution and [the visualization] is not showing what you expect it to show*." These debugging techniques focus on the latter, supporting the refinement of the user's mental model through exploration of both the data and state. To enable inspection of state and the behavior of changes over time, we incorporate three elements: a *timeline* of signals, *in situ annotations* of relevant encodings, and a *dynamic data table*. In the following sections, we describe the design and backing rationale for each of these debugging techniques.

5.1. The Signal Timeline and Replay

The *timeline* (Fig. 1f) lists every user-defined signal in specification order. Signal updates are represented as colored cells in the



```
{ "name": "xMax", "init": 1.6,
  "streams": [{
    "type": "xDelta",
    "expr": "xMax + (xMax-xMin)*xDelta/800"
  }]
}
```

Figure 3: *The overview, timeline, and signal annotations after performing interactions. (a) The overview provides insight into different interaction patterns. (b) Stepping within a pulse allows users to see intermediate states of an interaction. The second scatterplot shows a brush representing the* new `brush_start` *and old* `brush_end`*. (c) Dependencies are shown as red outlines on hover. (d) Signal annotations overlay the visualization, with fill color encoding temporality: from darkest (past), through red (current), to lightest (future).*

timeline, arranged into columns corresponding to reactive updates (pulses). The current signal value is displayed on the far right; mouse hover expands the contents and displays any scale transforms used to define the signal. As users interact with the visualization, signal values update and populate new columns in the timeline. By default, cell widths are automatically adjusted so all pulses are visible. An *overview* (Fig. 1e) summarizes pulse activity over time, with bar heights encoding the number of signal updates on a given pulse. The overview exposes patterns in the recorded interaction (Fig. 3a), and brushing zooms the timeline to show only pulses within the selected range.

Hovering over a cell displays a tooltip of the signal value in the overview to enable rapid comparison. Hovering also exposes the dependencies a signal update relies on — cells are outlined in red to illustrate which dependency values are used, and icons are shown beside dependency names in case the corresponding cell is not visible (Fig. 3c). Keyboard navigation allows users to move up and down, to understand the propagation of signal values within the same pulse (Fig. 3b), or left and right to identify a particular pulse which exhibited faulty behavior. The selected cell is indicated with dark green, with other signal values used by this state in light green.

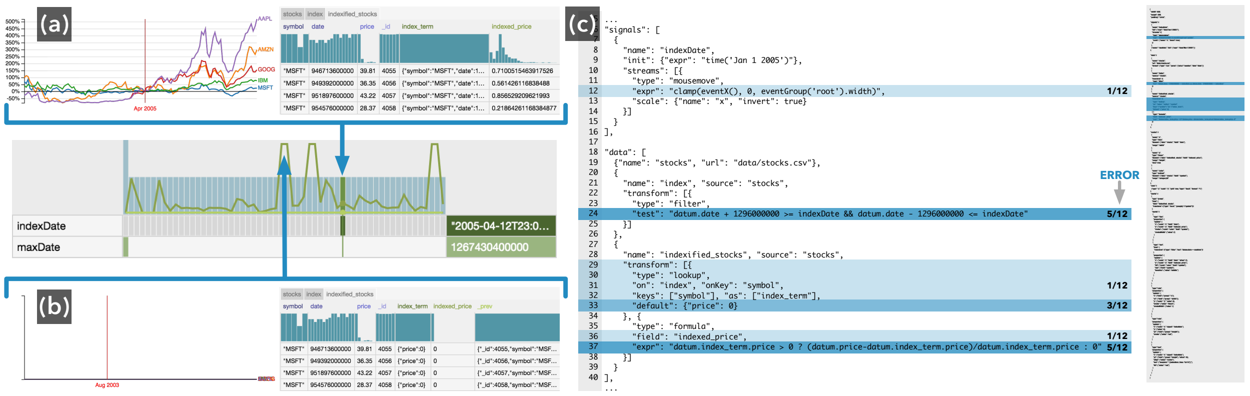
Users can select a cell in the timeline to *rewind* the visualization to an earlier state. Each time user interaction triggers a signal to update, the system records the new value and pulse number. Replay is enabled by setting the signal values for the desired pulse and re-rendering the visualization as if it were a new pulse in the specification. During replay, interaction is disabled to prevent new events from being added to the timeline mid-stream.

Rationale. The timeline provides users introspection into the heart of the interaction logic — signals — and is designed to reify the two-step reactive update process. As a result, pulses populate the timeline from top to bottom, and hovering over a particular cell can reveal if an older value was used for a dependency listed later. Early prototypes took this one step further. Pulse propagation was more salient as each cell in the timeline was marginally offset, producing a "cascade" or "waterfall" effect. This design required more space to encode the same information and made coarse navigation difficult. It was only meaningful to navigate left or right (i.e., backwards or forwards in time). As a result, locating a faulty pulse required users to step through every intermediate state of other pulses. In contrast, by condensing pulses into columns, users can quickly move back and forth across the timeline and only deep dive into the intermediate states of pulses of interest.

The timeline also maintains the level of abstraction provided by signals. For example, the particular low-level input events that trigger a reactive update are not identified. When such low-level events are required for debugging erroneous event selectors, users can define additional signals as needed that only capture the `event.type` that triggers them, and track them via the timeline and overview. Similarly, although Vega's internal dataflow dependency graph can be readily visualized, the timeline only surfaces dependency information for the particular cell a user hovers over. Helper signals automatically generated by Vega are also hidden from view. Together, these reflect the findings of our formative study: users were overwhelmed by details of Vega's execution pipeline, and found them to be tangential to the debugging at hand.

5.2. In-Situ Annotations

When users pause interaction recording, either explicitly (Fig. 1d) or by *rewinding* to an earlier state, a number of on-demand annotations become available to inspect the visualization state in-situ. The specification is analyzed to extract all scaled visual encoding rules for each mark. Mousing over the visualization performs a hit test against the underlying scenegraph to find an intersecting mark or group. If a mark is not found, then the user's cursor is over a group's background; the tooltip displays the cursor's coordinates relative to the group, along with any spatial scales used to encode the group's children (Fig. 1c). If a mark is found, its visual encoding rules are shown in addition to the coordinates.



stocks | index | indexified_stocks

symbol	date	price	_id	index_term	indexed_price
"MSFT"	946713600000	39.81	4055	{"symbol":"MSFT","date":1...	0.7100515463917526
"MSFT"	949392000000	36.35	4056	{"symbol":"MSFT","date":1...	0.561426116838488
"MSFT"	951897600000	43.22	4057	{"symbol":"MSFT","date":1...	0.856529209621993
"MSFT"	954576000000	28.37	4058	{"symbol":"MSFT","date":1...	0.21864261168384877

stocks | index | indexified_stocks

symbol	date	price	_id	index_term	indexed_price	_prev
"MSFT"	946713600000	39.81	4055	{"price":0}	0	{"_id":4055,"symbol":"MSF...
"MSFT"	949392000000	36.35	4056	{"price":0}	0	{"_id":4056,"symbol":"MSF...
"MSFT"	951897600000	43.22	4057	{"price":0}	0	{"_id":4057,"symbol":"MSF...
"MSFT"	954576000000	28.37	4058	{"price":0}	0	{"_id":4058,"symbol":"MSF...

```
...
"signals": [
  {
    "name": "indexDate",
    "init": {"expr": "time('Jan 1 2005')"},
    "streams": [{
      "type": "mousemove",
      "expr": "clamp(eventX(), 0, eventGroup('root').width)",                1/12
      "scale": {"name": "x", "invert": true}
    }]
  }
],

"data": [
  {"name": "stocks", "url": "data/stocks.csv"},
  {
    "name": "index", "source": "stocks",
    "transform": [{
      "type": "filter",
      "test": "datum.date + 1296000000 >= indexDate && datum.date - 1296000000 <= indexDate"   5/12
    }]
  },
  {
    "name": "indexified_stocks", "source": "stocks",
    "transform": [{
      "type": "lookup",
      "on": "index", "onKey": "symbol",                                     1/12
      "keys": ["symbol"], "as": ["index_term"],
      "default": {"price": 0}                                               3/12
    }, {
      "type": "formula",
      "field": "indexed_price",                                             1/12
      "expr": "datum.index_term.price > 0 ? (datum.price-datum.index_term.price)/datum.index_term.price : 0"   5/12
    }]
  }
],
...
```

Figure 4: *(a) An index chart interactively normalizes stock price time-series, (b) but a data transformation error zeros out the indexed price, flatlining the chart. (c) An excerpt of the specification shows the distribution of lines identified by participants as the source of the error.*

Mousing over a signal value in the timeline that duck-types to coordinates (i.e., an object with x and y properties), displays all signal updates as *signal annotations* on the visualization. The current point is denoted with a white stroke, and the fill color encodes temporality — older points are darker and lighter points occur further forward in the future (Fig. 3d). By default, signal annotations are only shown when hovering over the timeline; however, users can choose to have them drawn in real-time as interactions are recorded.

Rationale. Scale transforms are a common visual encoding operation, but can grow complex under a nested scenegraph model such as Vega's. For example, scales defined within nested group marks can shadow scales with the same name at higher levels. Generalizing an interaction technique requires invoking an inverse scale transform, to move from pixel to data values, but identifying the correct scale to use can be error-prone. Vega's scenegraph can be easily visualized but would still require a user to manually map its tree structure to the resultant visualization. Instead, our in-situ annotations make inspection of the scenegraph a direct manipulation operation. So as not to conflict with user-defined interactions, these annotations only appear when interaction recording is paused.

5.3. Dynamic Data Tables

Dynamic tables (Fig. 1h) display each specification-defined dataset and its attributes. The goal of the data tables is to give users a rapid, high-level sense of the backing data. Tables initially show only the first ten rows, which can be extended on-demand. This sample data allows users to review the attributes of each dataset and histograms summarize the distribution of each attribute at the given timestamp. Selecting a bar highlights corresponding values in the data table.

The data tables update automatically as the user interacts with the visualization to immediately depict changes in the distributions of data properties. While inspecting the table, a time series of the *variability* of each data property is shown in the overview (Fig. 1g). Mousing over the name of an attribute shows only the corresponding time series. The variability is calculated as follows, where bin'_i is the number of values in bin i of the histogram at the current state and bin_i is for the previous state: $\sum_{i \in bins} |bin'_i - bin_i|$. The variability for static attributes is a flat line along the bottom of the overview.

Rationale. In the formative study, one participant noted that users "*have this expectation about data... [that] is kind of unspoken and pretty hard to debug.*" Interactive visualization undoubtedly exacerbates this problem, as signals can further parameterize data transformations. By displaying the output data values of each dataset (i.e., after all transforms have been evaluated), our dynamic data tables narrow the gulf of evaluation [HHN85]. Moreover, the overview is augmented with dataset variability information to help users map the effect of signal updates to changes in the datasets. The current calculation for the variability detects large shifts in the distribution of data, instead of individual property values, in order to better highlight surprising changes. As with the timeline, datasets internal to Vega are hidden from view.

5.4. Linked Highlighting of the Specification

Users can select the name of a signal or data property in order to highlight all occurrences of that name in the specification in order to view the behavior in context. If the specification is not currently visible, it will be displayed alongside the current view. This linking allows users to rapidly trace variables to the original specification.

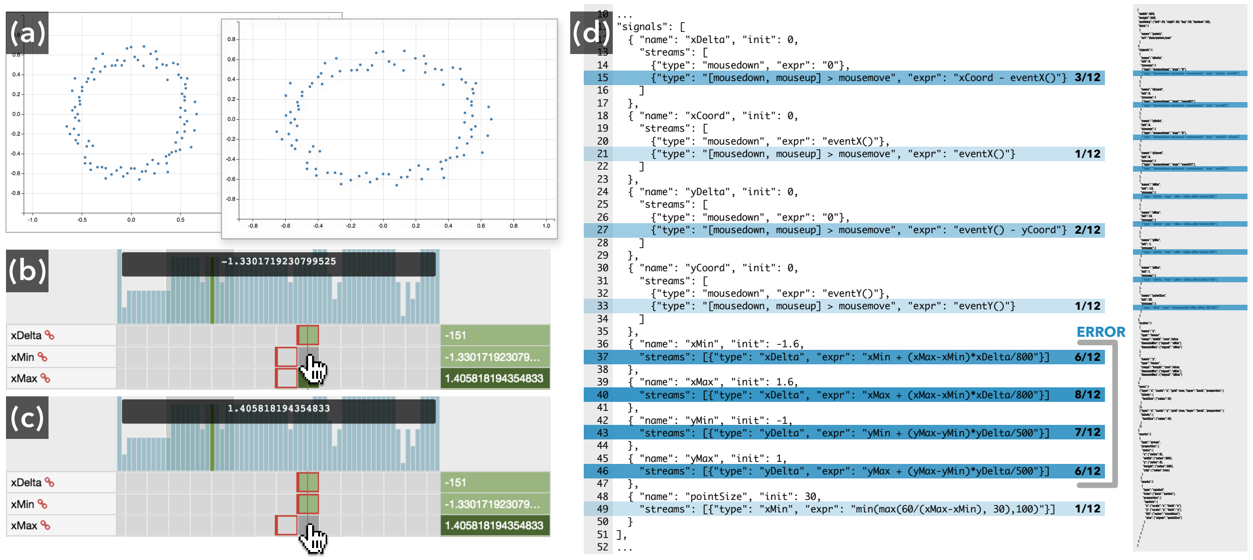
6. Evaluation: Debugging Faulty Visualizations

We conducted a study of how first-time Vega users utilize these debugging techniques to assess faulty specifications. Across a set of real-world errors, we examined participants' debugging strategies and their interactions with our visual debugging techniques.

Participants. We recruited 12 first-time Vega users (8 male, 4 female), all with prior experience analyzing data and creating visualizations. Participant ages ranged from 23 to 42 (mean 27.5, s.d. 5.14). All were either graduate (11) or postdoctoral (1) students at the University of Washington. Each study session lasted about 90 minutes; each participant received a $15 gift card as compensation.

Protocol. Prior to the study, we asked each participant to review the Vega beginner's tutorial[†]. We began the study with an additional Reactive Vega tutorial, including an introduction of the visual debugging techniques. Participants were provided with a reference

[†] https://github.com/vega/vega/wiki/Tutorial



```
...
"signals": [
  { "name": "xDelta", "init": 0,
    "streams": [
      {"type": "mousedown", "expr": "0"},
      {"type": "[mousedown, mouseup] > mousemove", "expr": "xCoord - eventX()"} 3/12
    ]
  },
  { "name": "xCoord", "init": 0,
    "streams": [
      {"type": "mousedown", "expr": "eventX()"},
      {"type": "[mousedown, mouseup] > mousemove", "expr": "eventX()"}           1/12
    ]
  },
  { "name": "yDelta", "init": 0,
    "streams": [
      {"type": "mousedown", "expr": "0"},
      {"type": "[mousedown, mouseup] > mousemove", "expr": "eventY() - yCoord"} 2/12
    ]
  },
  { "name": "yCoord", "init": 0,
    "streams": [
      {"type": "mousedown", "expr": "eventY()"},
      {"type": "[mousedown, mouseup] > mousemove", "expr": "eventY()"}           1/12
    ]
  },
  { "name": "xMin", "init": -1.6,
    "streams": [{"type": "xDelta", "expr": "xMin + (xMax-xMin)*xDelta/800"}]    6/12
  },
  { "name": "xMax", "init": 1.6,
    "streams": [{"type": "xDelta", "expr": "xMax + (xMax-xMin)*xDelta/800"}]    8/12
  },
  { "name": "yMin", "init": -1,
    "streams": [{"type": "yDelta", "expr": "yMin + (yMax-yMin)*yDelta/500"}]    7/12
  },
  { "name": "yMax", "init": 1,
    "streams": [{"type": "yDelta", "expr": "yMax + (yMax-yMin)*yDelta/500"}]    6/12
  },
  { "name": "pointSize", "init": 30,
    "streams": [{"type": "xMin", "expr": "min(max(60/(xMax-xMin), 30),100)"}]  1/12
  }
],
...
```

Figure 5: *(a) As the user pans the visualization, the axes stretch and distort the plot. This occurs due to an interdependency in the definition of the signals responsible for setting the range of the scale. (b) The min signal uses the old values of both min and max to compute its new value, whereas (c) the max signal uses the new min value and the old max value, thus causing the difference to drift. (d) An excerpt of the specification indicates the problematic lines and shows the distribution of lines identified by participants as the source of the error.*

sheet containing the names and descriptions of each technique (included in supplementary material). At the start of each task, we oriented participants with a brief explanation of the visualization and its intended functionality. Participants could then view the specification and interact with the visualization and debugging techniques. Participants were asked to diagnose the specific bug and identify one or more lines in the specification that cause the error.

Participants completed three tasks, each with an unfamiliar specification. We used existing specifications rather than having participants craft new visualizations from scratch to focus the evaluation on known debugging challenges. Each specification was based on a real-world error encountered by Vega users and developers. The selected errors represented a range of breakdowns, covering data transformation, interaction logic, and visual encodings, respectively. Tasks were ordered by increasing conceptual difficulty and emphasize different parts of the system. Detailed descriptions of each task are provided in the following sections.

The decision to use existing specifications ensured that each participant encountered the same set of errors and facilitated comparison across participants. Participants' unfamiliarity with Vega provided a conservative test of our debugging techniques, as participants could not rely on prior experience to inform the debugging process. As described in §4, the previous debugging strategy required user familiarity with the Vega system internals, which is not necessary when authoring visualizations. Given that most users lack the necessary familiarity to understand the system internals, the previous debugging process is not representative of the behavior of real-world users and is not a fair comparison for our expected use case. Manual exploration of the internal Vega structure is more low level than the abstraction used when writing specifications and thus less fit for general debugging scenarios. Future work is required to assess how these visual debugging techniques will be employed in real-world development processes by experienced users.

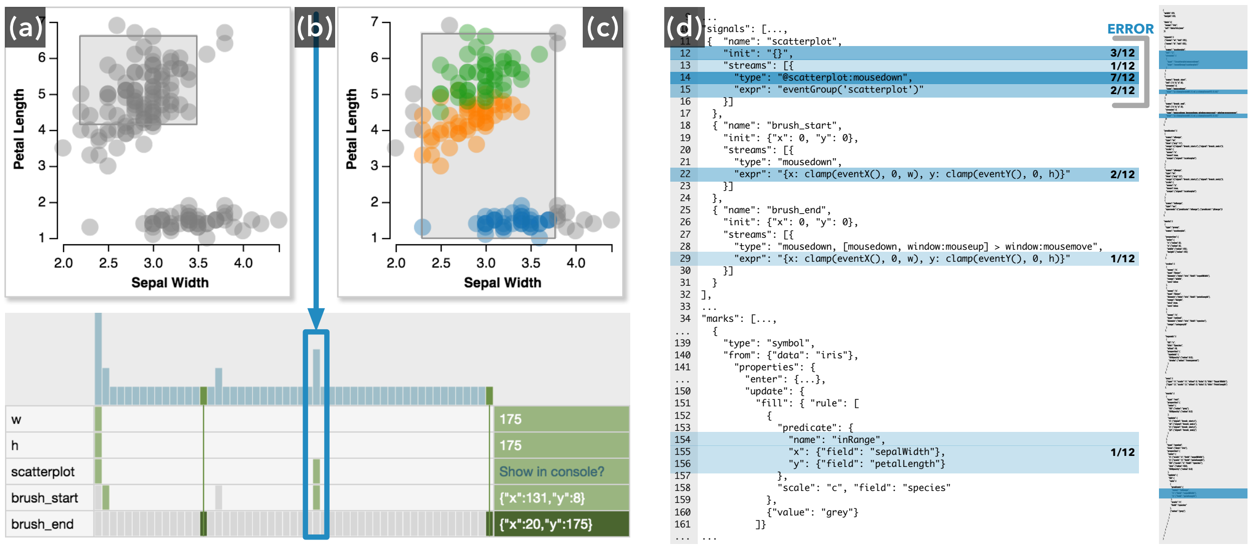
Data Collection. We used a think-aloud protocol throughout the study. Audio and screen recordings were captured for later review. At the end of each task, participants completed a brief survey in which they identified faulty lines of the specification and explained the reasoning for their choice. Participants additionally provided Likert ratings of the usefulness of each debugging technique. At the end of the study, participants ranked the debugging features and provided written impressions of the debugging experience.

Analysis. We assessed participant accuracy by checking if they correctly identified lines in the specification related to the error. We examined the average rating of each technique for each task and assessed the utility of techniques for different types of errors.

6.1. Data Transformation Errors: Index Chart

An index chart of stock prices renormalizes the data relative to a mouse-selected time point. At certain dates in the visualization, each line flatlines due to an erroneous data transformation that incorrectly filters the backing dataset (Fig. 4b). The filter uses a constant to specify a range with the same month and year as the index point, but the constant incorrectly excludes some points due to a slight time offset between the data and index point. The error can be resolved by using Vega's date support to compare the month and year. Participants had 15 minutes for this task.

Five participants (42%) correctly identified the exact line causing the error. All remaining participants correctly identified dependent lines that are corrupted by the faulty data filter (Fig. 4c). Participants identified nine distinct lines from the specification (out of 145). Participants started by identifying dates at which the visualization flatlines. By replaying to those points in the timeline, all participants verified that the signal value for the index point was capturing a logical date. Participants switched to the data table to compare attributes across states and observed that during the er-



```
...
"signals": [...,
  {  "name": "scatterplot",
     "init": "{}",                                                          3/12
     "streams": [{                                                          1/12
       "type": "@scatterplot:mousedown",                                    7/12
       "expr": "eventGroup('scatterplot')"                                  2/12
     }]
  },
  { "name": "brush_start",
    "init": {"x": 0, "y": 0},
    "streams": [{
      "type": "mousedown",
      "expr": "{x: clamp(eventX(), 0, w), y: clamp(eventY(), 0, h)}"        2/12
    }]
  },
  { "name": "brush_end",
    "init": {"x": 0, "y": 0},
    "streams": [{
      "type": "mousedown, [mousedown, window:mouseup] > window:mousemove",
      "expr": "{x: clamp(eventX(), 0, w), y: clamp(eventY(), 0, h)}"        1/12
    }]
  }
],
...
"marks": [...,
  {
    "type": "symbol",
    "from": {"data": "iris"},
      "properties": {
        "enter": {...},
        "update": {
          "fill": { "rule": [
            {
              "predicate": {
                "name": "inRange",
                "x": {"field": "sepalWidth"},                               1/12
                "y": {"field": "petalLength"}
              },
              "scale": "c", "field": "species"
            },
            {"value": "grey"}
          ]}
...
```

Figure 6: *Scale definitions are extracted from the `scatterplot` signal for the color encoding. The signal is initialized as an empty object, causing (a) the brush to display but leaving the points grey. When a point is clicked, (b) `scatterplot` is defined and (c) the brush works correctly for all future interactions. (d) An excerpt of the specification displays the distribution of lines identified as the source of the error.*

ror condition, the `indexed_price` was always zero. Participants linked back to the specification to identify dependencies and select candidate lines. Participants rated *replay* and the *data table* most highly (Fig. 7). The data table is essential for identifying corrupted data values from the faulty filter transformation. Replay is crucial for isolating the error states of the visualization.

6.2. Interaction Logic Errors: Panning a Scatterplot

A scatterplot supports panning via mouse drag. Over repeated panning actions, the aspect ratio of the plot distorts (Fig. 5a). Panning is implemented as a set of signals defining the minimum and maximum domain values for each axis. The error occurs due to a mutual dependency between these signals: the minimum signal uses the old minimum and maximum to compute the new value (Fig. 5b), whereas the maximum signal uses the *new* minimum value and *old* maximum value (Fig. 5c). To resolve the error requires a re-design of the specification to remove the mutual dependency in the interaction logic. Participants had 20 minutes for this task.

Eight participants (67%) correctly identified the minimum/maximum signals as the source of the error. The remaining participants identified either immediate upstream or downstream dependencies of the erroneous signals (Fig. 5d). Participants identified nine distinct lines from the specification (out of 136). Participants began debugging by panning the plot and forming hypotheses about the behavior of the error. In testing each hypothesis, participants often reset the timeline to only view the most recent signal updates. Once participants observed the distortion, they used the timeline to compare the signal behaviors. To assess the relationships between signals, some participants used the *dependency* markers to determine how the signal values propagated whereas others attempted to glean these relationships from the specification. Due to participants' lack of familiarity with the Vega syntax, reading the specification alone in the short time frame was a challenge. Participants noticed that many signals computed the difference between the minimum and maximum to represent the visible range, and noted that the size of this range should not be changing during the panning interaction. Participants thus identified signals utilizing this computation as candidate lines. Users rated the *dependencies* and *timeline* most highly for this task, as they revealed the relationships between signal values and the underlying interaction logic (Fig. 7).

6.3. Visual Encoding Errors: Brushing a Scatterplot

A scatterplot enables brushing to highlight points: points within the brush extents should have their fill color updated. The pixel values of the brush extents are run through scale inversions to determine a selection over data attributes. However, the brushing interaction does not always highlight points when the visualization is first parsed (Fig. 6a). The error occurs because the `scatterplot` signal (which represents a group mark containing the plot) is needed to find the appropriate scale to invert the pixel-level brush extents, but is initialized as an empty object that is only set on `mouse-down` events. However, these events do not correctly propagate if the user performs a `mousedown` on the background, as the enclosing group element has no fill color (an idiosyncrasy inherited from Scalable Vector Graphics). If the `mousedown` occurs over any of the plotting symbols, which do have a fill color, the event fires and `scatterplot` is accordingly set (Fig. 6b), enabling all future brushing actions to work appropriately (Fig. 6c). This example is a simplification of a breakdown that can occur in scatterplot matrices. Participants had 15 minutes for this task.

Nine participants (75%) correctly identified the `scatterplot` signal as the source of the error (Fig. 6d). The remaining participants incorrectly selected lines associated with the brush signal and the fill color encoding. Participants identified eleven distinct lines from the specification (out of 176). Two participants implemented

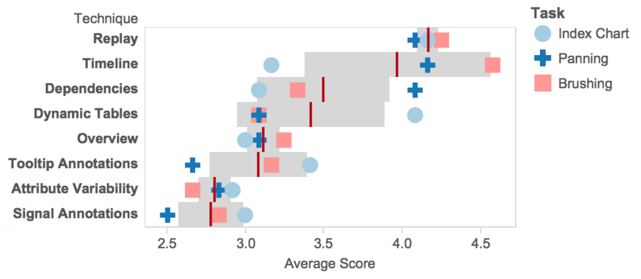


Figure 7: *Average ratings for each debugging technique, by task (shape) and overall (lines), with one standard deviation (gray).*

a partial fix by changing the definition of the scatterplot signal to update on `mouseover` instead of `mousedown`. While this solution causes the brush to correctly color points, it does not correctly address the problem of event propagation described above.

Participants began by attempting to reliably reproduce the erratic brushing behavior. Once the conditions of the behavior were determined, participants examined the timeline to compare the signals across working and faulty brushing runs. Participants observed that when a mark was selected, the scatterplot signal was set in the timeline to the appropriate scope. By selecting the scatterplot signal in the timeline, users highlighted its use in the specification in order to identify the corresponding specification lines. Consequently, the *timeline* received the highest ratings for this task (Fig. 7).

7. Discussion and Future Work

For each task, the majority of participants were successful in either precisely identifying erroneous specification lines or detecting lines directly related to the error. Despite being first-time users, participants accurately identified erroneous lines for faulty panning (67%) and brushing (75%) interactions. Three participants even attempted partial fixes (1 panning, 2 brushing) — an encouraging result given their lack of familiarity with Vega. In only 15-20 minutes, these participants were able to observe, diagnose, and start experimenting with solutions to the error in an unfamiliar specification in an unfamiliar environment. For the index chart, 42% of participants correctly identified the problematic line, with the remaining participants identifying dependent lines corrupted by the error. As participants used our debugging techniques to conceptually home in on an unfamiliar problem, we consider this a promising result.

Figure 7 plots participant ratings for each debugging technique. Of particular note is that the utility of each technique is highly dependent on the type of error — for example, *dynamic tables* rated highly for the index chart, which featured a data transformation error, whereas the *timeline* was rated poorly as the interaction required only a single signal. In order to understand the complex dependencies within the panning example, the *dependencies* on the timeline were much more salient. On average, the combination of *timeline* and *replay* techniques were deemed universally useful for assessing program state and observing relevant changes (Fig. 7). One participant noted that "*the combination of the timeline, replay, automatic text highlight, and dependencies makes for a pretty useful and smooth debugging experience*."

The remaining techniques (*overview*, *tooltip*, *attribute variability*, and *signal annotations*) were rated lower on average as each technique was less effective at surfacing information relevant to the debugging tasks. The *tooltip* had the largest spread of average ratings, and was particularly useful in debugging the index chart by allowing users to inspect the encoding of the broken state and track the underlying error to the backing dataset. The *attribute variability* was designed to support quick identification of data changes, but was often overlooked by participants. One user noted that their low rating suggests that the system should "*promote its appearance more*." Currently, users must explicitly select the debugging technique they wish to use, which requires them to know what information would be most useful. Further development of these techniques might examine how to automatically surface relevant details with less user intervention. Additional static analysis, or new higher-level specifications, could help the system better understand the semantics of interactions (e.g., do signals define point or range selections?) and automatically surface appropriate techniques.

The replay technique currently updates the visualization by setting the signal values of the previous state and re-rendering the visualization as if it were a new pulse in the execution. However, this functionality assumes a consistent definition of the set of signals, limiting support for hot-swapping changes in the specification. Future work should examine what additional information should be recorded to support replay of interactions across specification changes. Though low-level input events are abstracted into signal definitions for easier debugging by users, such events may be necessary to support replay when signal definitions have changed.

In the evaluation, one participant explained that "*I would have loved a way to use the visualization essentially as an editor to modify the specification (and then see those changes update the viz in real time)*." Lyra [SH14] provides an interactive environment for visualization design via direct manipulation, but does not yet support authoring interactions. Our timeline visualizes the propagation of events to the interaction logic, but may be too low level for an interactive development environment like Lyra. By shifting the focus of the signal annotations from a summary of all events, to an indication of the current state, the annotations could support better debugging of interaction sequences in-situ. Replay could then support playback and refinement of interaction sequences to enable authoring of interactions in an interactive design environment.

Interaction techniques are crucial for exploring and understanding visualizations, but are often difficult to author and debug. Vega's reactive semantics encapsulate the bulk of the interaction logic, providing a meaningful entry point for the debugging process. We contribute a set of visual debugging techniques that allow users to probe the state, visualize relationships, and inspect state transitions over time. The three tasks in the user evaluation demonstrate data transformation, interaction logic, and encoding errors that arise during the design of interactive visualizations. The evaluation demonstrates how the proposed visual debugging techniques can be used by novice users to accurately identify and understand these errors and better support their debugging needs.

Acknowledgements

This work was supported by Intel Big Data ISTC, the Moore Foundation, DARPA XDATA, and a SAP Stanford Graduate Fellowship.

References

[BBKE13] Burg B., Bailey R., Ko A. J., Ernst M. D.: Interactive record/replay for web application debugging. In *Proceedings of the 26th annual ACM symposium on User interface software and technology* (2013), ACM, pp. 473–484. 2

[BCC*13] Bainomugisha E., Carreton A. L., Cutsem T. v., Mostinckx S., Meuter W. d.: A survey on reactive programming. *ACM Computing Surveys (CSUR) 45*, 4 (2013), 52. 2

[BH09] Bostock M., Heer J.: Protovis: A graphical toolkit for visualization. *IEEE Trans. Visualization & Comp. Graphics 15*, 6 (2009), 1121–1128. 3

[BOH11] Bostock M., Ogievetsky V., Heer J.: D3: Data-Driven Documents. *IEEE Trans. Visualization & Comp. Graphics 17*, 12 (2011), 2301–2309. 3

[CC13] Czaplicki E., Chong S.: Asynchronous functional reactive programming for guis. In *Proc. ACM SIGPLAN* (2013), ACM, pp. 411–422. 2

[CL08] Cottam J., Lumsdaine A.: Stencil: a conceptual model for representation and interaction. In *Information Visualisation* (2008), IEEE, pp. 51–56. 2

[DFS02] Demetrescu C., Finocchi I., Stasko J. T.: Specifying algorithm visualizations: Interesting events or state mapping? In *Software Visualization*. Springer, 2002, pp. 16–30. 3

[GKM82] Graham S. L., Kessler P. B., Mckusick M. K.: Gprof: A call graph execution profiler. In *ACM Sigplan Notices* (1982), vol. 17, ACM, pp. 120–126. 3

[Goo15a] Google: JavaScript Memory Profiling. https://developer.chrome.com/devtools/docs/javascript-memory-profiling, December 2015. 3

[Goo15b] Google: Profiling JavaScript Performance. https://developer.chrome.com/devtools/docs/cpu-profiling, December 2015. 3

[Gre15] Gregg B.: Flame Graphs. http://www.brendangregg.com/flamegraphs.html, December 2015. 3

[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. 2

[HHN85] Hutchins E. L., Hollan J. D., Norman D. A.: Direct manipulation interfaces. *Human-Computer Interaction 1*, 4 (1985), 311–338. 6

[HS12] Heer J., Shneiderman B.: Interactive dynamics for visual analysis. *Queue 10*, 2 (2012), 30. 1

[KL15] Kelleher C., Levkowitz H.: Reactive data visualizations. In *IS&T/SPIE Electronic Imaging* (2015), International Society for Optics and Photonics, pp. 93970N–93970N. 2

[KM04] Ko A. J., Myers B. A.: Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (2004), ACM, pp. 151–158. 2

[LBM14] Lieber T., Brandt J. R., Miller R. C.: Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems* (2014), ACM, pp. 2481–2490. 2

[MGB*09] Meyerovich L. A., Guha A., Baskin J., Cooper G. H., Greenberg M., Bromfield A., Krishnamurthi S.: Flapjax: a programming language for ajax applications. In *ACM SIGPLAN Notices* (2009), vol. 44, ACM, pp. 1–20. 2

[MHHH15] Moritz D., Halperin D., Howe B., Heer J.: Perfopticon: Visual query analysis for distributed databases. *Computer Graphics Forum (Proc. EuroVis) 34*, 3 (2015). 3

[Mye91] Myers B. A.: Separating application code from toolkits: eliminating the spaghetti of call-backs. In *Proc. ACM UIST* (1991), ACM, pp. 211–220. 1

[OM09] Oney S., Myers B.: Firecrystal: Understanding interactive behaviors in dynamic web pages. In *Visual Languages and Human-Centric Computing, 2009. VL/HCC 2009. IEEE Symposium on* (2009), IEEE, pp. 105–108. 2

[PSCO09] Pike W. A., Stasko J., Chang R., O'Connell T. A.: The science of interaction. *Information Visualization 8*, 4 (2009), 263–274. 1

[SH14] Satyanarayan A., Heer J.: Lyra: An interactive visualization design environment. *Computer Graphics Forum (Proc. EuroVis)* (2014). 9

[SRHH15] Satyanarayan A., Russell R., Hoffswell J., Heer J.: Reactive vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2015). 1, 3

[SWH14] Satyanarayan A., Wongsuphasawat K., Heer J.: Declarative interaction design for data visualization. In *ACM User Interface Software & Technology (UIST)* (2014). 1, 2, 3

[Vic12] Victor B.: Inventing on Principle. https://vimeo.com/36579366, January 2012. 2

[WTH02] Wan Z., Taha W., Hudak P.: Event-driven FRP. In *Practical Aspects of Declarative Languages*. Springer, 2002, pp. 155–172. 1, 2, 3