Abstract—Managing complex data flows and update patterns is one of the most difficult challenges in interactive data visualization. For example, constructing interactive visualizations with multiple linked views can be a daunting task. Functional reactive programming provides approaches for declaratively specifying data dependency graphs and maintaining them automatically. We argue that functional reactive programming is an appropriate and effective abstraction for interactive data visualization. We demonstrate the effectiveness of our proposed approach in several visualization examples including multiple linked views. We also provide a catalog of reusable reactive visualization components.

Keywords—multiple linked views; interaction techniques; information visualization;

I. INTRODUCTION

Constructing interactive visualizations is a complex task. The task becomes even more complex when multiple visualizations are presented and linked together through interactions. The issue at the core of interactive visualization and linked views is management of complex data flows and update patterns. Even with the wealth of visualization toolkits and libraries that exist today, there is a need for an abstraction that addresses these core issues. The contribution of this paper is a novel approach for developing reusable interactive visualization components using concepts from functional reactive programming.

II. RELATED WORK

The first attempt at a systematic formalization of data visualization was Jacques Bertin’s “Semiology of Graphics” [1]. In this work, Bertin relates data types to visual marks and channels in a coherent system that takes visual perception into account. Bertin’s work has influenced many future theoretical
underpinnings of visualization, including Leland Wilkinson’s “Grammar of Graphics” [2] and Jock Mackinlay’s APT (A Presentation Tool) system [3], which led to the development of the commercial visualization package Tableau [4].

Interactions within data visualization environments have been well studied. Becker et al. investigated brushing in scatter plots [5]. Shneiderman et al. explored dynamic queries in general and how these operations fit into a larger context of visual information seeking [6]. Ward introduced a visualization system based on multiple linked views with direct manipulation techniques including brushing and linking [7]. Anselin discussed how interactive visualization systems with linked views can be applied to Geographic Information Systems [8]. Yi et al. conducted a thorough survey of existing taxonomies for visualization and interactions and developed a set of generalized classes of interactions for visualization [9].

Much work has been done regarding interactive visualization of data cubes. Stolle et al. introduced a formalism for defining multi-scale visualizations of data cubes throughout their work on the Polaris system [10] [11] [12]. Cuzzocrea et al. surveyed the area of data cube visualization in depth [13]. Mansmann coined the term “Visual OLAP”, framed it as a fundamentally new paradigm for exploring multidimensional aggregates [14]. Scotch et al. developed and evaluated SOVAT, a Spatial OLAP visualization and analysis tool applied to community health assessments [15] [16]. Several interactive “Big Data” visualization systems have been introduced [17] [18] [19]. Techapichetvanich et al. explored how visualization interactions pertain to data cubes in particular [20]. Sifer et al. introduced a visual interface utilizing coordinated dimension hierarchies for OLAP cubes [21].

Interactions within data visualization environments have been well studied. Becker et al. investigated brushing in scatter plots [5]. Shneiderman et al. explored dynamic queries in general and how these operations fit into a larger context of visual information seeking [6]. Ward introduced a visualization system based on multiple linked views with direct manipulation techniques including brushing and linking [7]. Anselin discussed how interactive visualization systems with linked views can be applied to Geographic Information Systems [8]. Yi et al. conducted a thorough survey of existing taxonomies for visualization and interactions and developed a set of generalized classes of interactions for visualization [9].

The World Wide Web has evolved to become a full fledged application development platform. HTML5 is the latest set of standards and APIs (Application Programming Interfaces) from the World Wide Web Consortium that define the capabilities of modern Web browsers [21]. HTML5 applications are able to run across multiple platforms (albeit requiring some effort from developers). HTML5 has eclipsed Java Applets and Flash in fulfilling the dream of “write once, run anywhere”. HTML5 contains three graphics technologies that can support interactive Web-based visualizations: Canvas, SVG (Scalable Vector Graphics), and WebGL.

D3.js is a flexible and powerful visualization library that uses SVG and has a strong community of users [22]. D3 at its core is a DOM manipulation library with heavy use of functional programming. D3 allows concise declarative statements to define the core logic of visualizations. D3 provides additional APIs for performing common visualization tasks such as defining and using scales, generating labeled axes, and computing layouts from graphs and trees. D3 is at the center of a vibrant developer ecosystem and has seen wide adoption in industry. There are plentiful examples of D3.js usage for creating visualizations. Many supporting libraries have been created including NVD3 reusable charts, Chart.js for composing visualization elements, Crossfilter.js for interactive multidimensional filtering, and DC.js for multiple linked views.

Interactive data visualizations can be linked together such that interactions in one visualization cause updates in another visualization. This technique is referred to as “multiple linked views” [23] and “brushing and linking” [24] [25]. This technique overcomes limitations of single visualizations by supporting exploration of the data through interaction. More information can be presented to the user with multiple linked views as compared to static visualizations. Dynamic queries, a technique related to multiple linked views, allow the user to interactively define query parameters used for generating the input data for a visualization [6].

The Model View Controller (MVC) architecture is a long standing best practice for organizing complex applications [26]. The MVC architecture was first introduced as part of the Smalltalk-80 system for building user interfaces [27], and has been used extensively for Web application development [28]. Several authors describe how the MVC architecture can be applied to visualizations with multiple linked views [29], [30], [31], [32].

Functional reactive programming provides techniques for declaratively specifying reactive data dependency graphs [33]. Elliott et al. applied functional reactive programming to animation [34]. Hudak et al. applied functional reactive programming to robotics [35]. Data flow is a concept related to functional reactive programming in which developers can specify directed graphs of data transformations [36]. The KNIME data analysis environment uses a data flow model as its primary abstraction [37].

III. REACTIVE MODELS

In this section we describe a novel way to combine elements of functional reactive programming with the Model View Controller (MVC) paradigm to create what we call reactive models. These reactive models can serve as a foundation for reusable interactive visualization components. This approach overcomes limitations of traditional MVC frameworks, and is simpler than using a full blown functional reactive programming framework.

The Model View Controller paradigm can be combined with functional reactive programming to enable straightforward creation of reactive systems based on data flow graphs. In this section we introduce a novel and simple approach for implementing reactive models.
The Model in the Model View Controller (MVC) paradigm is responsible for:

- managing the state of the application,
- allowing the Controller to change the state of the application, and
- notifying the view when the state of the application changes.

One simple and widely used method for structuring a Model is as a set of key-value pairs [28]. This kind of model can fulfill the all of the responsibilities of a Model with three methods:

- `set(key, value)` Set the value for a given key.
- `get(key)` Get the value for a given key.
- `on(key, callback)` Add a change listener for a given key. Here, `callback` is a function that will be invoked synchronously when the value for the given key is changed.

We first discuss a simple model with only `set` and `get` methods (SimplestModel), then discuss a more complex version that also includes `on` (SimpleModel), then finally introduce a reactive model that includes the `when` operator from functional reactive programming (Model).

### A. Simplest Model

```javascript
SimpleModel = λ()
values = { }
return
set : λ(key, value) values[key] = value
get : λ(key) return values[key]
```

The above pseudocode implements a key-value model that has only `set` and `get` methods. Line 1 defines the constructor function, SimplestModel, which will return a new object that has `set` and `get` methods. Line 2 defines a private variable called `values` that will contain the key-value mapping. Lines 3, 4 define the `set` and `get` methods, which store and retrieve values from the internal `values` object. Here's an example of how SimplestModel might be used.

```javascript
mySimplestModel = SimpleModel()
mySimplestModel.set('x', 5)  // Causes line 3 to log 5
mySimplestModel.get('x')    // Evaluates to 5
```

### B. Simple Model

Here is a version of the model that implements the `on` method as well:

```javascript
SimpleModel = λ()
values = { }
callbacks = { }
return
on : λ(key, callback)
if callbacks[key] == NIL
  callbacks[key] = []
callbacks[key].push(callback)
set : λ(key, value)
values[key] = value
if callbacks[key] ≠ NIL
  for callback in callbacks[key]
    callback()
get : λ(key) return values[key]
```

The above version includes an additional private variable, `callbacks`, which is an object whose keys are property names and whose values are arrays of callback functions. The `on` method defined starting at line 5 adds the given callback to the list of callbacks for the given key (and creates the list if it does not yet exist). The `set` method has been modified to invoke the callback functions associated with the given key when the value for that key is changed. Here is an example of how the `on` method can be used.

```javascript
mySimpleModel = SimpleModel()
mySimpleModel.on('x', λ())
log(mySimpleModel.get('x'))
)
mySimpleModel.set('x', 5)  // Causes line 3 to log 5
mySimpleModel.set('x', 6)  // Causes line 3 to log 6
```

### C. Functional Reactive Model

For complex applications such as interactive visualizations, managing propagation of changes can quickly become complex. For this reason, modular visualization environments based on data flow have become popular [38]. A data flow graph defines a directed acyclic graph of data dependencies. The data flow model is amenable to construction of visual programming languages [39]. While many systems consider data flow as a means to construct data transformation pipelines, the concept also applies to building reactive systems that manage change propagation throughout an application or subsystem in response to user interactions or other events [34].

To provide a solid foundation for dynamic visualization systems, the Model should be able function in the context of data dependency graphs. Developers should be able to declaratively specify data dependencies, and change propagation should be automatically managed. The `when` operator from functional reactive programming propagates changes from one or more reactive functions (such as is found in the JavaScript libraries Bacon.js and RXJS).

Our Model implementation can be extended with a `when` operator that enables construction of data dependency graphs. This operator will become a foundation for building dynamic interactive visualizations. Since `when` is superior to `on` in that it handles change propagation intelligently, in this final version...
on is not exposed in the public Model API. Adding \texttt{when} depends on having some utility functions available, \texttt{debounce} and \texttt{allAreDefined}.

```
1  debounce = \lambda (callback)
2       queued = FALSE
3   return \lambda (
4     if queued == FALSE
5       queued = TRUE
6       run(\lambda ()
7       queued = FALSE
8     callback()
9   )
```

The \texttt{debounce(callback)} function returns a function that, when invoked one or more times in a single codepath, will queue the given \texttt{callback} function to execute only once on the next tick of the event loop. This has the effect of collapsing multiple sequential calls into a single call. The returned function is referred to as the “debounced” function.

The \texttt{debounce} function defined starting on line 1 creates a closure with a boolean variable \texttt{queued} (instantiated on line 2) that keeps track of whether or not the \texttt{callback} function is currently queued to execute in the future. When the debounced function (defined starting on line 3) is called the first time, the condition on line 4 evaluates to \texttt{TRUE}. This causes \texttt{queued} to be set to \texttt{TRUE} (on line 5) and also causes the function defined starting on line 6 to be queued to run in the future. This uses the built-in function \texttt{run} that queues a function to execute on the next tick of the event loop.

When the debounced function is invoked multiple times in the same code path, the condition on line 4 evaluates to \texttt{false}, and nothing happens. When the current code path terminates and the queued function is invoked, \texttt{queued} is set to \texttt{false} (on line 7) and the \texttt{callback} function is invoked.

```
1  allAreDefined = \lambda (array)
2    for item \in array
3  if item == NIL
4    return FALSE
5  return TRUE
```

The function \texttt{allAreDefined(array)} checks if all values in the given \texttt{array} are defined. It does so by comparing each item in the array to the special value NIL. As soon as one item is found to be NIL, the function returns \texttt{false} (on line 4). If all items have been checked and none are found to be NIL, the the function returns \texttt{true} (on line 5). We are now ready to define our model that includes the \texttt{when} operator.

```
1  Model = \lambda ()
2  simpleModel = SimpleModel()
3  return
4    set : simpleModel.set
5    get : simpleModel.get
6  when : \lambda (dependencies, fn)
7    callFn = debounce(\lambda ()
8        args = dependencies.map(simpleModel.get)
9    if allAreDefined(args)
10      apply(fn, args)
11   )
12  callFn()
13  for key \in dependencies
14    simpleModel.on(key, callFn)
```

### IV. REACTIVE VISUALIZATIONS

In this section we discuss how our reactive model construction can serve as a foundation for interactive visualizations. We also show how reactive models can be linked together at a higher level to form interactive visualizations with linked views. Interactive visualizations must respond to changes made by users such as resizing of the display, changes in the data driving the visualization, changes in configuration, and updates from other visualizations in a linked view context. We introduce reusable reactive visualization components, shown in table [I] and show how they can be composed to form interactive visualizations and linked views.

#### A. Visualization Primitives

Consider visualizations such as the bar chart, line chart, stacked area chart, parallel coordinates and choropleth map. These visualizations share many underlying primitives such as scales, axes, margins and labels (22). Interactive forms of these visualizations also share interaction techniques for selecting visual marks such as rectangular brushing, hovering, clicking, panning and zooming. These visualization primitives can be encapsulated as data dependency subgraphs within reactive models which we call \texttt{components}.

Property names serve as the common elements between components. Below is a listing of the property names used in one or more components and a description of each:

- \texttt{data} The relation to be visualized.
- \texttt{xAttribute} The attribute within the data relation mapping to the X Axis.
- \texttt{xScale} The scale (domain and range) for the X axis.
- \texttt{xScaleType} Determines whether \texttt{xScale} is a linear, logarithmic or ordinal scale.
- \texttt{xAxis} The visible X axis (tick marks and labels).
- \texttt{xAxisLabel} The X axis label text.
- \texttt{yAttribute} The attribute within the data relation mapping to the Y Axis.
- \texttt{yScale} The scale (domain and range) for the Y axis.
- \texttt{yScaleType} Determines whether \texttt{yScale} is a linear, logarithmic or ordinal scale.
Table I shows a listing of components that can be combined to easily generate a foundation for a variety of interactive visualizations. These components encapsulate reactive data dependency subgraphs that implement the visualization primitives necessary for interactive visualizations. Figure 3 shows how several of these components can be assembled to create a general-purpose reactive bar chart.

![Figure 3](image-url)

**Figure 3.** A visualization of the Iris data set [40] using linked views, powered by reactive visualization components. Brushing to select records in the scatter plot causes the selected data to be aggregated and displayed in the bar chart.

V. Conclusion

In this paper we introduce a novel way to combine elements of functional reactive programming with the Model View...
Controller (MVC) paradigm to create what we call reactive models. These reactive models allow developers to declaratively specify data dependency graphs. This kind of abstraction is well suited for developing interactive visualizations because it drastically simplifies management of complex data flows and update patterns.

We introduced a set of reusable data dependency subgraphs for interactive visualization we call components. These components encapsulate reactive flows involving visualization primitives such as margins, scales and axes. We demonstrated how these components can be composed for generating a reusable bar chart. We also demonstrated how these components can be used for assembling visualizations with multiple linked views.

VI. Future Work

Future directions for this work will focus on developing a catalog of reusable visualization components, coupling the data to currently available public data sources, developing visualization-centric user interfaces and collaboration.

So far we have applied our technique to bar charts and scatter plots only, however we intend to also support the following visualizations:

- Table - Work on this has been done linking data to a dynamic HTML table (see figure ??).
- Color Legend
- Line Chart
- Pie Chart
- Choropleth Map
- Parallel Coordinates
- Heatmap
- Stacked Bar Chart
- Stacked Area Chart
- Streamgraph
- TreeMap
- Force Directed Graph Layout

On the visualizations where it is appropriate, we plan to develop reusable components for the following interaction techniques:

- Brushing - Dragging to interactively define a selection rectangle.
- Picking - Selecting a single visual mark by clicking or tapping on it.
- Details-on-demand - A pattern of linked view composition in which an overview visualization can be used as navigation for a detailed view. This is a fundamental concept in interactive information visualization [43].

We hope to develop a user interface for quickly assembling reusable components together based on graph drawing. This user interface would show the data dependency subgraphs shown in the diagrams of this paper, but they would be dynamic and editable. Another direction we have started working on is integrating external user interface components such as traditional list selections, drop down menus, and radio buttons. One example of this concept is shown in figure 5 which shows how reactive models can provide reactive HTML tables that can be linked with interactive visualizations. This would help in constructing intuitive user interfaces for manipulating the configuration of visualizations.

An open source proof of concept implementation of our reactive visualization approach is available on GitHub. These projects contain the source code that generated figures 3 and 5. The GitHub pages also link to live demos you can try out yourself. The reactive model implementation is available at github.com/curran/model, and the reactive visualization components are available at https://github.com/curran/reactivis.

ACKNOWLEDGMENT

The authors would like to thank Professor Jesse Heines of University of Massachusetts and Professor Rosane Minghim of University of Sao Paulo, Sao Carlos - SP, Brazil for giving constructive feedback throughout the development of this work.

REFERENCES
