A DeVIL-ish Approach to Inconsistency in Interactive Visualizations

Yifan Wu
UC Berkeley
yifanwu@berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@berkeley.edu

Eugene Wu
Columbia University
ewu@cs.columbia.edu

1. INTRODUCTION

Declarative languages have a long tradition in both the database systems and data visualization communities, separating specifications from implementations. In databases, declarative languages like SQL shield application programmers from changes to physical and logical properties like disk layouts, indexes and schema changes. In data visualization, declarative languages like Polaris, ggplot2 and Vega shield visualization programmers from variations in rendering, including screen layout, resolution, and color schemes. Declarative languages have been considered a foundational step forward in both communities.

To date, most of the work on declarative languages in the visualization community has focused on describing static visualizations: the positions, shapes and colors of graphical marks to be rendered. Interaction has been accepted as a key aspect of modern visualization systems [7], yet work on introducing declarativity to interactive visualizations is still in its infancy. Recent systems such as Reactive Vega [9, 10] have introduced higher-level specifications for interactions in a functional reactive programming (FRP) model. Synergistically, we are studying the benefits of a **De**clarative **V**isual **I**nteraction **L**anguage (DeVIL) that enables the specification of interactive visualizations in a declarative framework inspired by languages like SQL and Polaris. In particular, DeVIL presents a unified relational model for interactive visualization applications, and provides a basis to analyze aspects of interaction such as performance, correctness, and expressiveness.

In this paper, we focus on a specific benefit provided by DeVIL: managing the consistency of interactive visualizations in the face of inherent asynchrony and reordering of events in modern data visualizations. We begin by illustrating how the quest for increased interactivity and scale in data visualization can lead to unintended, confusing or undesirable user experiences (inconsistencies). We show how our declarative approach naturally lets us borrow consistency strategies from the database literature, freeing programmers from the need to build ad-hoc mechanisms to achieve consistency in their applications.

HILDA'16, June 26 2016, San Francisco, CA, USA

ISBN 978-1-4503-4207-0/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2939502.2939517

2. DATA MODEL

DeVIL takes a relational approach towards interactive data visualization and models nearly all logical representations of the visualization—input events, base data, marks (e.g., SVG elements), and even the rendered pixels—as database tables. In other words, data computation, visually encoding data values into marks, layout, and rendering are *logically* encapsulated within the DeVIL data model. We emphasize the term *logically* because the physical representation may not be practical to implement. For instance the pixels are significantly more efficient when represented as a rasterized image, however we can logically model them as a database table for analysis purposes. Although a full treatment of the data model is beyond the scope of this paper, we will outline the key concepts in order to ground the examples in the subsequent sections.

At a high level, static visualizations are expressed as view definitions over a set of base data tables (Figure 1). These view definitions logically describe how to aggregate and transform the data, how to map those records into a set of marks in the *Marks table*, and how to render the visualization into a table of pixels shown to the user. Complex computations such as layout algorithms and rasterization are modeled as table UDFs.

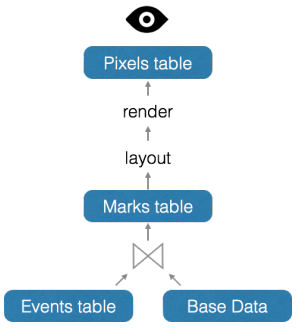


Figure 1: High-level DeVIL data model.

Interactions are modeled as a join between an *Events table* and the base data. User interactions (e.g., drag) are composed of a stream of event records (e.g., mousemove) that are appended to the *Events table*. Since the visualization is modeled as a set of view definitions, each event record triggers updates to all of the views, including the pixels table. The updated state of the pixel table solely depends

on the previous state of the database along with the newly appended event.

The *Events*, *Marks* and *Pixels* tables also represent different levels of abstractions in the data model. Section 4 will show cases where these abstractions are used when resolving inconsistencies in the visualization.

This table-centric model provides numerous benefits by allowing us to borrow analysis and execution techniques from the database literature [12]. For instance, we can use notions of versioning and database lineage to easily facilitate time-travel capabilities such as undo and replay as well as linked interactions such as cross-filtering [11]. In addition, we can now view the visualization through the lenses of materialized view maintenance, and explore automatic ways to incrementally update the visualization. In contrast, existing visualization developers have to implement these features by hand. This is not only error-prone, but also, as we will explore in Section 3, exposes the visualization to undesirable consistency issues that are difficulty to manage and debug. Finally, modeling the visualization state as tables allows us to both borrow and adapt existing concurrency analysis, and enables powerful policy descriptions as simple logical statements over these tables.

3. SOURCES OF REORDERING

The data model sketched in Section 2 can capture the semantics of the visualization and interactions, but there are many sources of non-deterministic reordering in any interactive system: asynchronous execution, software components that introduce reordering, and concurrent interactions. They could cause undesired behaviors in interactive visualizations, and we will illustrate with a few examples.

We start with a fully synchronous example that induces a correct ordering at the expense of significant interaction delays. The user drags their mouse to select bars in a summary bar chart (e.g., sales by month). This 1) resizes the selection box 2) highlights the selected bars and 3) renders the disaggregated records of the highlighted bars in a scatterplot visualization. We assume that the selection box and highlights are nearly instantaneous to update, whereas the cost to compute and render the linked visualization is high (perhaps due to network latency or database access).

Figure 2a shows a time plot of a synchronous execution of this example, where the bottom is most recent. The user's drag motion generates two mouse move events. `1:move` represents a move event in time T1, which is sent to the backend in the same timestep. However, the processing takes a long time (denoted by two timesteps at the backend), thus the entire interface, including dispatching mouse movements, is blocked until visualization is updated in T3. Only then, can the next `2:move` event from the interaction occur in T4. Under this execution model, long backend processing time drastically reduces the interactivity of the visualization application and can lead to a poor interactive experience, which has been shown in literature to reduce user data analytics performance [5]. In the following examples, we will explore different ways that applications could introduce asynchrony as a mechanism to increase concurrency as well as improve interaction response times. We find that this can introduce unintended side effects in the update order of the visualization.

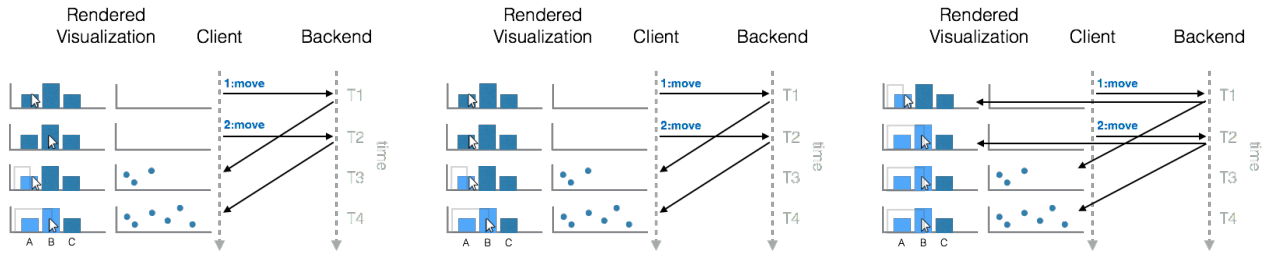
Asynchronous Event Dispatch, Synchronous Visualization Updates: Event-driven programming models like Javascript and its various frameworks encourage asynchronous event handling as a way to ensure responsiveness to user input. However asynchrony is not a panacea. For instance, Figure 2b illustrates the effects of asynchronously dispatching events to the backend system. This execution model allows users to continue interacting with the visualization while the backend processes earlier events. But in this case, each event/response pair is handled synchronously, so the user will not see any visual changes for the `1:move` event until it is fully computed in T3.

External Sources of Reordering: Our next example illustrates how asynchronous event dispatch described above, along with non-deterministic backend software components can lead to unintended reorderings. For instance, visualization systems are commonly designed as a two tiered architecture, where a client (e.g., browser) renders the visualization and translates user interactions (e.g., the move events of a drag interaction) into a sequence of queries that are sent to a backend database system. The client then renders the query results. Although database systems guarantee serializability (e.g., transactions are executed equivalently to *some* serial ordering), they are allowed to reorder the transactions in arbitrary ways. Figure 3 shows how this can cause the updates for the `2:move` to arrive and render in T3, while the earlier `1:move` events is rendered later in T4.

Asynchrony Within an Interaction: A common way to improve interactivity is to introduce concurrency by decomposing the visualization update into smaller updates for each component. For instance, the mouse movement updates three components of the visualization—the size of the selection box, the color of the selected bars, and the contents of the scatterplot. Separate requests may be executed asynchronously for each of these components, so that some parts of the visualization can remain responsive. For example, Figure 2c decouples the updates to the bar chart and the scatter plot, so that selection and highlighting is reflected very quickly, whereas the scatterplot is allowed to take more time. In this way, the user can see *what* they are selecting in real time, although the detailed scatterplot view may take longer to render.

Although this is a common way to improve visualization interactivity, the asynchrony can lead to surprising effects in the visualization. For instance, T3 in Figure 2c shows bars `A` and `B` highlighted, however the scatterplot only shows data for bar `A`—components of the same visualization are showing results from different subsets of the base data. This can lead the user towards incorrect conclusions, so the developer *may* consider these visualization components *inconsistent*.

Concurrency Between Interactions: The asynchrony described above is an instance of concurrency within a single interaction. Similar reorderings are possible under concurrent interactions—for example, two users in a collaborative interface, or a single user using multiple input devices (e.g., hand tracking). Consider a new developer creating an interactive visualization for sales data (Figure 4). She first creates a selectable bar chart of sales by month (`month`) that updates the chart of sales in SF and NYC (`city`). She then copies and modifies `month` to create a selectable bar chart of sales by hour (`hour`). Unfortunately, in this configuration, each selectable chart independently updates `city`. For example, if the user selects bar `A` in `month`, then `city` shows



(a) Fully synchronous execution model blocks interaction inputs until visualization is updated.

(b) Asynchronous processing does not block inputs, but synchronous execution blocks visualization updates.

(c) Asynchronous processing and visualization updates improve interactivity but introduce unintended reorderings.

Figure 2: Time plots of visualization updates to the selection box, highlighted bars, and scatterplot based on user mousemove events.

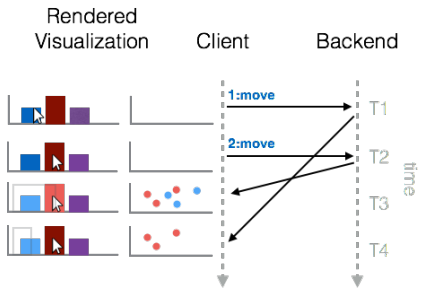


Figure 3: Time plot of reordering due to nondeterministic processing in the backend database.

SF and NYC sales for bar `A`, regardless of selections in `hour`.

Even in an idealized setting where updates are instantaneous (Figure 4), the concurrent interactions in `month` and `hour` can interleave the mouse move events. This can cause `city` to show data corresponding to `month`'s selected data in one instant (e.g., T1 and T3) and switch to `hour`'s selected data in the next instant (e.g., T2 and T4). The interleaved visualization updates due to the interleaved interaction events can cause users tremendous confusion.

Note that the reordering in this example came from the interleaving of the events from the two concurrent interactions. There are some obvious ways to address this issue, for example, by disallowing concurrent interactions altogether, while other policies may only allow concurrent interactions as long as they don't interfere with each other. We will discuss policies to specify these semantics in the next Section.

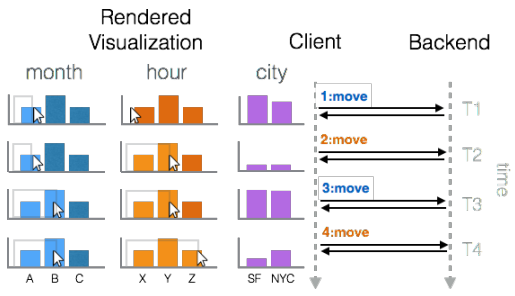


Figure 4: Time plot of interleaved events between two concurrent interactions: undesirable reorderings are possible even if processing is instantaneous.

This section has described reorderings due to different forms of asynchrony. Many of these re-orderings result in inconsistent and, we argue, incorrect visualization states that can confuse or mislead users. Going forward we will treat them in two separate categories: individual interaction and interleaved interactions, due to the resolution semantics which will be discussed. In the following sections, we will see other cases where the reorderings can be declaratively resolved. The power of the DeVIL approach is that it provides a framework to reason about, and express what the consistency semantics should be.

4. DEALING WITH REORDERINGS

So far we've seen that a combination of concurrency and asynchrony can be used to improve visualization interactivity, however at the cost of potentially undesirable result reorderings. This is not a new problem. Event-driven programming models like Javascript and its various frameworks encourage asynchronous event handling to ensure responsiveness to input. But a side-effect of that responsiveness is complexity: the order of event handling can be hard to predict and control. There have been numerous efforts to deal with this problem from the front-end community which we will discuss in related work section. This section discusses how DeVIL makes it possible to detect potential conflicts from difference sources on various levels, and provide a framework to reason about reordering and declarative ways to specify and ensure correct behavior.

4.1 Database Conflict Analysis

Traditionally in databases, acceptable concurrency is defined by a serializability guarantee, ensuring that the queries are executed by a transaction as if in isolation. A common serializability guarantee is conflict serializability, which ensures that the reads and writes of all transactions occur in an order equivalent to some serial ordering of the transactions [4]. In Section 2 we described how the UI is modeled as *Marks* and *Pixels* tables, which are view definitions over the *Events* and *Base Data* tables. In the case of interleavings due to concurrent interactions, the interactions share the same logical screen, so there may be write conflicts in the edits to the *Marks*, *Pixels* or other shared tables in the data model. Given this setup, we may address concurrency between interactions *modeling an interaction, and all of the view updates due to its events, as a transaction.* We could then apply standard database locking to the materialized

tables that represent visualization views.

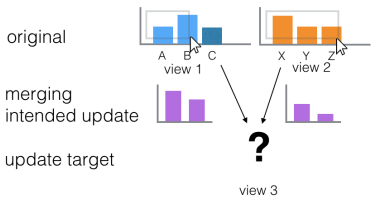


Figure 5: Update diagram of two interactions updating the same view with different values.

To make things concrete, let's take a look at the example in Figure 5. A brushing interaction on the blue chart (view 1) triggers an update, shown as the purple chart along the arrow, to a secondary view (view 3), depicted as the question mark. The orange chart (view 2) tries to update view 3 with a different chart. If these updates are modeled as updates to individual pixels in the pixels table, then write-write conflicts become apparent.

Now that we see how write conflicts could arise in DeVIL and how conflict analysis could prevent potentially erroneous updates to the UI, we will demonstrate scenarios where this analysis can automatically identify "satisfactory" reorderings in terms of visualization semantics, and allow more concurrency than a strict serial interaction ordering. We will see that different acceptable interation semantics can be expressed by varying the granularity of the analysis (table, record, and cell level) and the tables for which we apply lock based analysis (*Pixels* and *Marks*).

Table Level Locking Naively, the entire pixel table could be locked during the execution of an interaction (including both the user events and the computations needed to perform the interaction query). This simple method yields correct output without any potential anomalies but severely decreases the interactivity, which negates the point of this whole exercise, which is to increase interactivity via asynchrony.

Record Level Locking Let's consider reducing the locking granularity to the record level for the *Pixel table*. As seen in Figure 6, selecting the blue bars in view 1 updates view 3, and selecting the orange bars in view 2 updates view 4. Since there are no record level conflict, the interactions are allowed to happen concurrently, which is an improvement from the previous table level locking scheme.

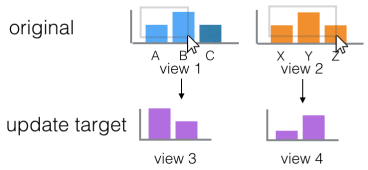


Figure 6: Update diagram of two interactions updating different non-overlapping views, without record level conflict.

Cell Level Locking Pushing for even further concurrency, we can study cell level locking on the *Marks table*. In Figure 7 where the same mark is modified by two interactions, but one modifies the color, the other the transparency. These two interactions do not have any conflicts on the *Marks table* cell level but will have a conflict on records in the *Pixel table*, so in this case we are allowing more concurrent interactions to happen.

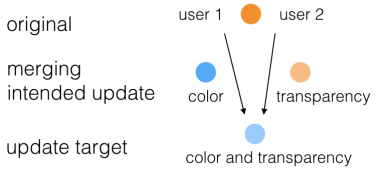


Figure 7: Update diagram of two interactions updating color and transparency of a single mark respectively, without cell level conflict

Figure 8 is an example of cell level read-write conflict. User 1 (cursor) is brushing to select bars `A` and `B` in the view (say in a linked selection visualization), while user 2 (hand pointer) concurrently attempts to drag bar `A`—these forms of interaction are commonplace on touch-based visualizations such as Tableau Vizable. If the bar is moved out of User 1's selection region, then it can unintentionally modify the results of User 1's selection. As mentioned before in Section 2, the User 1's selection can be logically viewed as a join between the user's selection region and the bars in the visualization based on their overlap. Since evaluating this join requires comparing the positions of the bars with the selection region, the join obtains a read lock on the mark position attributes. Thus, when the drag interaction attempts to update bar A's position, it cannot acquire the write lock until User 1's interaction is completed.

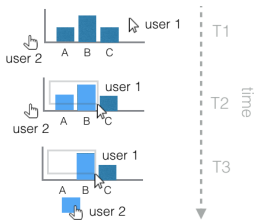


Figure 8: Scenario where user 1 selects bar `A` and `B` , user 2 drags bar `A` , creating a cell level read-write conflict on the position of bar `A` .

Our application of conflict analysis introduces the interesting issue of abort due to deadlocks. It is not clear how to unwind an ongoing interaction in a user friendly manner and the issue merits deeper investigation.

While database conflict analysis at the read/write level provide a simple and useful mechanism to work with reorderings, it does not always capture application semantics intelligently and only helps with interleaved interactions. As a result, we first introduce Merge Functions to allow orders that conflict but are semantically acceptable, which apply when conflict analysis is too conservative. We then introduce Interaction Constraints to prevent orders that do not conflict but lead to semantic inconsistencies which apply when conflict analysis is too liberal. Note that both of the mechanisms could deal with reorderings from single interaction as well as interleaved interactions.

4.2 Merge Functions

Traditional database conflict analysis may forbid desirable reorderings. In many cases, we can easily avoid unnecessary conflicts using visualization semantics. For example, in Figure 9, the "update diagram", the selections `AB` in view 1 updates with the blue dots, and the selection `XYZ` to the right in view 2 updates with the orange dots, which partially overlaps with the blue dots. Under record level locking on *Pixels table*, this scenario would conflict, since both interactions are writing to the pixels in the overlapped area, with different colors. However, in practice, we rarely see concurrency limitations due to these conflicts, suggesting that there are ways to avoid these issues.

In this subsection we will describe merge functions, which resolve conflicts between pairs of Pixel (or Marks, or data) tables. We will see that a large class of techniques used in the wild can be reduced to applications of merge functions at different tales in the DeVIL data model.

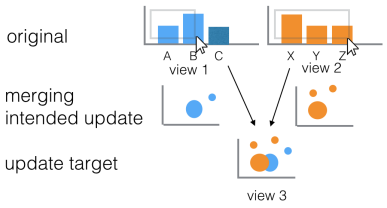


Figure 9: Update diagram of two concurrent interactions that "conflict" but could be correct under painter's algorithm semantics (described below).

Merging on the Pixel Layer One common technique in rendering from computer graphics is alpha composition [8], which exposes arithmetic for arbitrary composition between objects. For example, we could make two marks transparent, preserving information about the marks' position and colors. The visualization developer applies the transformation on the overlapping pixels, as seen in Figure 10.

Figure 10: Example of a merge function that combines the orange and blue points into the merged view via alpha composition.

Merging on the Mark Layer Following the same set up as the example in Figure 10, an alternative to pixel layer resolutions is on the mark layer, as demonstrated in Figure 11. Beginning with the first scenario, painter's algorithm (or priority fill), the merge function always takes the pixel assignment from one interaction consistently across all overlapping marks. The second example outcome in the same figure, the distortion function, merges the overlapping marks into a bigger mark. As for jittering, the merge function moves the two overlapping marks in the opposite direction by a certain number of pixels, and then apply painters algorithm as mentioned before, this way suggesting that there is more than one mark in the area. Finally, transparent overlay update both of the original marks' transparency, still revealing information about the fact that the marks are not a simple mark, but allowing two updates to happen concurrently.

Figure 11: Four examples of merge functions on mark layer.

Merging on the Interaction Layer Revisiting the conflict mentioned in Figure 5, we demonstrate a merge function to resolve the conflict in Figure 12. We could avoid overlapping pixels by repartitioning the updates to different parts of the UI: once a conflict is detected, dynamically update the layouts of the target views into spatial subsets of the original view. This is merging on the interaction layer because the merge function is applied to all the marks impacted by the respective interactions.

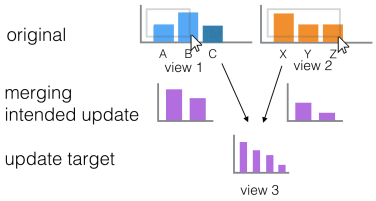


Figure 12: Update diagram of two concurrent interactions merged via spatial partitioning: shrinking the individual bar width to fit all marks within the original view.

Figure 13 is our final merge function example. The interaction in view 1 translates selections into a filter predicate which states that the attribute corresponding to the x-axis (`view1.x`) must be in the set `{A,B}`. The interaction additionally updates view 3 to show the subset of the base data table satisfying `view1.x` $\in$ `{A,B}`. View 2 involves a similar interaction to also update view 3. Rather than merge records directly, the merge function may instead compose the views' predicates using a conjunction (or disjunction) operator. In the figure, view 3 shows a scatterplot of the two base data records that satisfy both predicates.

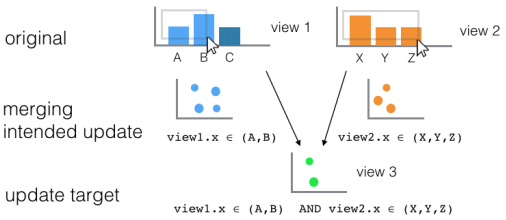


Figure 13: Update diagram of two concurrent interactions merged via predicate conjunctions: showing results that satisfy both of the predicates defined in the individual interactions.

This ability to work with different layers of the visualization objects, pixels, marks, and interactions, makes it possible to express a wide range of behaviors, which are otherwise difficult to implement in imperative models.

More broadly, merge functions are used in the database community as powerful tools for reasoning about reordering, as seen in commutative operators a la Sagas and CRDTs,

which were recently shown to generalize to any monotonic code, as in $Bloom^L$ and the CALM Theorem. Depending on the semantics of the merge function and lineage information, we could design a collection of merge functions for visualization use cases that are monotone, i.e. commutative, associative, and idempotent [1], and even perform compiler side optimization to discover monotonicity automatically. We intend to explore these in future work.

4.3 Interaction Constraints

While database conflict analysis and merge functions can express a wide range of consistency policies, there are still reasonable policies that cannot be expressed using the mechanisms introduced so far. In this section, we will give examples of how constraints in our data model offer a natural way to describe even richer consistency semantics. These interaction constraints are logical statements analogous to classic integrity constraints. Note in the examples below we use some syntactic sugar to abstract away implementation details, which merit a deeper discussion outside of this paper.

Multi-visualization Consistency Let us return to Figure 2c, where the asynchronous execution of each visualization component results in a visualization state that shows inconsistent data in T3. The bar chart shows that two bars are highlighted, however the scatterplot only shows the data for bar A. The designer may be concerned about the visualization user drawing wrong conclusions from the intermediate states, and assert that the input records from `basedata` that generated the marks in the bar chart and scatterplot (i.e., their lineage) should be the same, as shown in example snippet below.

```
lineage(barchart, basedata) == lineage(scatter, basedata)
```

Specific Concurrency Control A visualization designer is concerned that two interactions, though with no data level conflicts, when happening concurrently might mislead the user to draw wrong conclusions (perhaps due to implicit assumption connecting the two unrelated interactions). The designer could use the following constraint to prevent two interactions from happening simultaneously.

```
!concurrent(interaction1, interaction2)
```

Interaction Responsiveness Guarantee To prevent unresponsive UX in the case of an unexpectedly long running query, a designer may wish to kill long running interactions that are blocking.

```
seconds(now - current_interaction.begin) < 60
AND is_blocking(current_interaction)
```

The examples above only illustrate some scenarios and there are many other potentially useful policies. While we do not have space to detail the implementation of these policies, it's worth noting that the ability to access multiple layers of the data model and global state at any timestep allows us to declaratively describe and enforce invariants. One could implement these policies with an event driven imperative programming model, but it will require a lot of machinery to manage global states across time.

5. RELATED WORK

There are several recent approaches to improve design patterns for visualization and front-end programming. Rx [6] and React [2] are emblematic of event-driven functional reactive programming (E-FRP) languages that simplify the composition of asynchronous and event-based programs. Many of the concepts are similar to stream query operators, however their model is not explicitly relational, and tools for analyzing and addressing the inconsistency issues highlighted in the paper are lacking.

Reactive Vega [9] adopts an FRP execution model to interactive visualizations and introduces a high level data flow specification as well as a compact short-hand (Vega-lite) for expressing common visualization and interaction patterns. From the database community, FORWARD [3] models web applications as materialized view definitions over a nested relational model. However neither directions have focused on advanced tools to resolve inconsistencies in the visual interaction due to asynchrony and concurrency.

6. FUTURE WORK

DeVIL's unified data model enables direct application of conflict analysis tools from concurrency control, distributed systems, and constraint logic. In addition, its grounding in view definitions enables potential new work such as lineage support that can be leveraged when expressing interaction constraints. Further work is required to understand how the three proposed mechanisms can compose together. In addition, we are currently in the process of developing a DSL for expressing the policies categorized in this paper, and building a prototype system to evaluate our proposed solutions.

References

[1] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, 2011.

[2] Facebook. React. https://facebook.github.io/react/.

[3] Y. Fu, K. W. Ong, Y. Papakonstantinou, and M. Petropoulos. The sql-based all-declarative forward web application development framework. In *CIDR*, pages 69–78, 2011.

[4] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992.

[5] Z. Liu and J. Heer. The effects of interactive latency on exploratory visual analysis. *TVCG*, 2014.

[6] E. Meijer. Reactive extensions (rx): curing your asynchronous programming blues. In *SIGPLAN*, 2010.

[7] W. A. Pike, J. Stasko, R. Chang, and T. A. O'connell. The science of interaction. *IV*, 2009.

[8] T. Porter and T. Duff. Compositing digital images. In *ACM Siggraph Computer Graphics*, volume 18, pages 253–259. ACM, 1984.

[9] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. *TVCG*, 2016.

[10] A. Satyanarayan, K. Wongsuphasawat, and J. Heer. Declarative interaction design for data visualization. In *UIST*, 2014.

[11] C. Weaver. Cross-filtered views for multidimensional visual analysis. In *TVCG*, 2010.

[12] E. Wu, L. Battle, and S. R. Madden. The case for data visualization management systems. *VLDB*, 2014.