Libra+: Compositional Interaction for Data Visualization

Xu Luo, Yue Zhao, Bongshin Lee, Jean-Daniel Fekete, and Yunhai Wang

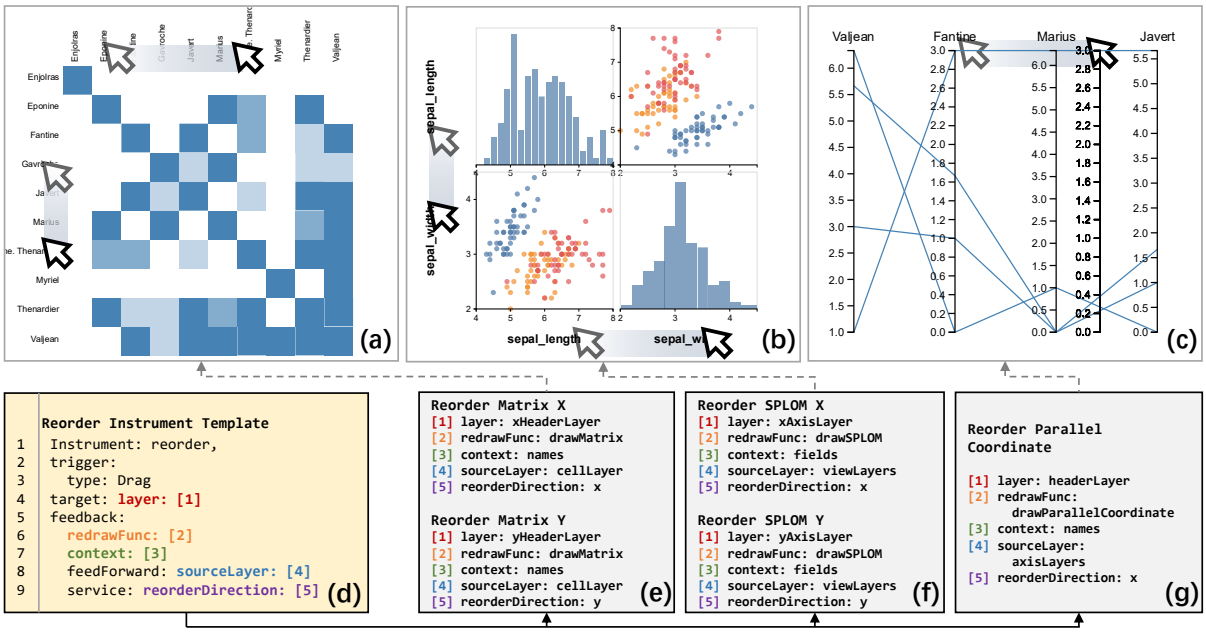


Fig. 1: Libra+ streamlines interaction design through the composition of parameterized instruments. A reordering instrument template (a) can be readily applied across diverse visualizations. By configuring the parameters shown in color, developers can compose multiple instruments to support different reordering behaviors. For example, combining two distinct instruments enables both X- and Y-axis reordering in a matrix plot (a, e) and a scatterplot matrix (b, f), whereas only a single configured instrument is needed for a parallel coordinates plot (c, g). Several additional instruments can be added to these visualizations.

Abstract—We present Libra+, a high-level conceptual model and library for interactive data visualization that supports the composition of primitive interactions into sophisticated ones. Libra+ formalizes interactions through a Trigger–Target–Feedback model, which specifies input events that initiate an interaction, the target on which it operates, and the resulting feedback. This model serves as a semantic bridge between high-level interaction intent and its modular, concrete implementation. To operationalize this approach, we define atomic instruments as reusable interaction units and introduce coordination strategies for composing them across graphical layers and views. By resolving input conflicts, supporting cross-layer synchronization, and enabling customizable feedback, Libra+ provides both a vocabulary for interaction design and a practical framework for building modular, extensible, and composable interactions for visualizations. We implement this model as an extension of Libra.js [32] and demonstrate that Libra+ can reproduce sophisticated interactive visualizations and generalize existing interactions to new contexts through a set of use cases. By enabling the parametrized composition of instruments, our unified interaction framework promotes modularity and extensibility in visualization interaction design.

Index Terms—Information visualization, interaction, model, composition

1 Introduction

Interactive data visualization enables analysts to explore complex datasets through direct manipulation and iterative analysis. Modern visualization systems support a small set of interacton techniques, including brushing, filtering, panning, and zooming, that allow users to focus on subsets of data, reveal patterns, and coordinate insights across multiple visual representations. As analytical workflows increasingly rely on such interactions, visualization systems must provide flexible mechanisms for specifying and enriching interactions.

While the Grammar of Graphics [27] unified the specification of visual encodings, enabling easy specification of complex visualizations, interaction design remains comparatively limited. Popular toolkits such as D3 provide a limited set of generic interactions easy to add to visualizations, such as pan, zoom, and hover, and recipes for a few more like point selection, but they lack underlying mechanisms or guidelines to build richer interactions. Hence, existing D3 visualizations rarely implement advanced interactions due to the excessive effort required and lack of reusability. Declarative grammars such as Vega-Lite [20] advanced the field by modeling interaction as a reactive programming problem and by introducing the selection abstraction. However, reactive programming addresses only a partial aspect of interaction; it

- *Xu Luo and Yunhai Wang are with Renmin University of China, Beijing, China. Email: {luoxu1998,wang.yh}@ruc.edu.cn.*
- *Yue Zhao is with Shandong Second Medical University, Weifang, Shandong, China. Email: zhaoyue@sdsmu.edu.cn.*
- *Bongshin Lee is with Yonsei University, Seoul, Republic of Korea. E-mail: b.lee@yonsei.ac.kr.*
- *Jean-Daniel Fekete is with Inria and Université Paris-Saclay. E-mail: jean-daniel.fekete@inria.fr.*
- *Yunhai Wang is the corresponding author.*

Manuscript received xx xxx. 202x; accepted xx xxx. 202x. Date of Publication xx xxx. 202x; date of current version xx xxx. 202x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.202x.xxxxxxx

often couples interaction management directly to specific visual marks, creating monolithic units that are difficult to extend and reuse across contexts. To address this, Libra [32] introduced interaction instruments that manage interaction events originating from visualization views, acting as mediators between users and the visual presentation. By decomposing interactions into modular components, Libra associates each instrument with specific graphical layers within views, such as the background, main, and transient layers, ensuring that visual feedback remains structurally separated from underlying data marks and instruments become more generic, lowering the complexity of adding advanced interactions to visualizations.

Despite these advances, managing interactions remains a challenge for developers. While Libra effectively separates interactions into internal components, making them more reusable and modular, the specification of new interactions remains low-level and verbose, which limits rapid authoring and systematic design exploration. Three key factors limit the modularity of Libra instruments: (1) competition between instruments for events, (2) lack of formal specification of the scope in which instruments operate, and (3) overly tight coupling between instruments and feedback. For example, when an instrument handles pan navigation and another handles brushing selection, coordination is needed to determine which instrument should take precedence when the user starts clicking. This coordination is not well addressed in existing systems, when it should be as simple and explicit as possible to the application programmer.

To address this issue, we present Libra+, a high-level model for interactive data visualization that treats complex interactions as compositions of simple ones in a systematic way. Each interaction is parameterized by three orthogonal aspects: *Trigger–Target–Feedback* (TTF), input event sequences that start the interaction (*Triggers*), scopes into which the interaction is triggered, including graphical layers (*Targets*), and service–transformer pipelines to provide feedback during the execution of an interaction (*Feedback*). This structural decomposition bridges high-level interaction intent and modular execution, allowing interactions to be represented in a more composable and reusable manner. Building on this model, we introduce a set of simple atomic instruments that serve as building blocks for sophisticated interactive behaviors. Complex interactions can be defined by composing these instruments with coordination rules. Libra+ encapsulates the relationships between input processing, affected visual layers, and resulting feedback.

In addition, we contribute to clarifying the vocabulary for interaction design by decoupling interaction intent from visual feedback. This separation addresses common ambiguities in which terms such as "brushing" and "linking" are often conflated with their visual outcomes, ultimately enabling clearer and more systematic descriptions of interactive behaviors.

We implement this model in a prototype[1] built on top of the Libra.js library [32], leveraging its layered architecture and service-based model. To evalute Libra+, we demonstrate its ability to reproduce sophisticated interaction techniques, generalize interaction templates across diverse contexts, and enrich interactions through the composition and extension of multiple instruments. Libra+ not only provides a consistent formal vocabulary for interaction design but also significantly enhances the modularity, reusability, and extensibility of interaction code.

2 Related Work and Background

In this section, we review models, specifications, and taxonomies of interaction in visualization, and provide a brief overivew of the Libra interaction model.

2.1 Interaction in Visualization: Models, Specifications, and Taxonomies

Models for Interactive Visualization. Interactive visualization has long been recognized as a key approach to exploratory data analysis. Early frameworks, such as Shneiderman's principles of direct manipulation [23], emphasized iterative exploration to reveal patterns and generate insight. These works highlighted two enduring ideas: interactions should support flexible exploration over diverse data subsets, and visualization systems should enable coordinated views.

Despite these foundational insights, most early models and taxonomies describe interactions at a high level without providing mechanisms for modular implementation or composition. Classical GUI models, such as Smalltalk's MVC [19], separate software into modular components, improving maintainability, but they often tie components to specific interaction behaviors and offer limited support for coordinating multiple overlapping inputs. Similarly, visualization-specific architectures, including the Visualization Reference Model [7] and Prefuse [14], formalize pipelines from data to view and propose design patterns for certain interactions. However, yet these solutions remain partial and largely application-specific.

Declarative grammars, such as the Grammar of Graphics [27] and Vega-Lite [20], formalize visual encodings and provide concise abstractions for reactive interactions via constructs such as selections. These approaches allow developers to describe interactive behaviors succinctly, but they primarily focus on what interactions accomplish rather than how they are structured or composed. Consequently, interaction definitions often remain monolithic, tightly coupled to specific visual marks, and difficult to reuse or extend across multiple charts.

Instrument-based models offer a more modular perspective. Instrumental interaction [2] provides a conceptual foundation for modeling human–tool mediation, and Libra [32] operationalizes this idea in visualization by decomposing interactions into *Interactors*, *Services*, and *Transformers*. This design separates user intent, computation, and feedback across layers, improving clarity and reusability. However, Libra does not explicitly provide mechanisms for coordinating multiple instruments when they compete for the same input or visual space. Our work builds on this foundation by introducing a structured semantic decomposition and explicit coordination strategies, enabling modular, composable, and conflict-aware interactions.

Interaction Specifications for Visualization. Early interactive visualizations were often implemented using event-based programming with callbacks, which quickly led to complex and tangled "spaghetti code" [18]. Libraries such as Protovis [4] and D3 [5] continue to rely on this approach for defining custom interactions. Developers must manually maintain state machines and coordinate overlapping calls, making interactions difficult to reuse, extend, or compose. One common solution is to encapsulate interactions within black-box objects. Toolkits like Prefuse [14], Improvise [26], VisDock [8], and DIVI [24] provide predefined techniques—such as selection, navigation, and annotation—to simplify reuse, but these often remain rigid and monolithic.

Higher-level frameworks adopt more declarative approaches to simplify interaction specification. Vega [21, 22] leverages functional reactive programming (FRP) [25] to treat input events as data streams, allowing developers to compose interactions while implicitly managing internal state. Vega-Lite [20] further introduces a grammar of interactions, using selection primitives to streamline interaction design. While these frameworks improve expressiveness, they do not fully expose feedback mechanisms, and extensions (e.g., signal recording [16]) are needed to customize or capture interaction effects. As a result, creating, extending, and reusing sophisticated interactions remains challenging.

Recent advancements, such as reactive widgets [13], further promote a modular philosophy that replaces monolithic designs with reusable, composable units linked through reactive programming and standard web events. However, while these frameworks primarily focus on the encapsulation and assembly of high-level views and widgets, Libra+ addresses a different scope: the internal decomposition of direct manipulation itself.

In contrast, Libra [32] offers a transparent framework for defining all stages of interaction, from input to feedback, in a modular and composable manner. Libra separates transient interactive elements from the underlying visualization by placing them on dedicated, synchronized layers. This design isolates feedback from persistent data marks, enabling interactions to operate independently while maintaining visual consistency. Furthermore, Libra generically supports feedforward by

[1] https://libra-plus.github.io/

leveraging undo/redo mechanisms to preview interaction outcomes and revert them if the action is not confirmed.

However, creating new interactions in Libra requires developers to work directly with all underlying components. Implementing sophisticated behaviors often involves low-level manipulation of multiple objects, which can make the process complex and cumbersome despite the modular architecture. Instead, Libra+ introduces a high-level interaction model that enables developers to systematically construct complex interactions by composing simple, atomic instruments.

Interaction Taxonomies and Vocabulary. Given the central role of interaction in data visualization, extensive prior research has sought to formalize it through taxonomies and conceptual vocabularies. For instance, Yi et al. [30] classify interaction techniques based on user intent, proposing categories such as select, explore, reconfigure, encode, abstract/elaborate, filter, and connect. Building on this, Heer and Shneiderman [15] delineate interactive dynamics essential for visual analysis, encompassing selection, navigation, filtering, and cross-view coordination. These classifications build upon foundational mechanisms for exploratory analysis, such as dynamic queries [1] and brushing and linking [3].

Despite their foundational value, existing taxonomies are primarily descriptive. They articulate the purpose of various interactions but offer little structural guidance on how these interactions can be modularized, reused, or composed. A major limitation is that colloquial terms like "brushing," "linking," "selection," and "filtering" frequently conflate three distinct dimensions: user intent, execution mechanism, and visual outcome. As Dimara and Perin [10] highlight, "what 'interaction' means in the context of visualization is ambiguous and confusing." This lack of standardization hinders the development of interactions as reusable building blocks, as analogous behaviors are often labeled disparately, while functionally distinct behaviors are grouped under same terms.

This ambiguity creates significant barriers to interaction composition. Current taxonomies fail to articulate how multiple concurrent interactions relate, how conflicts should be resolved when they share input modalities, or how their visual effects can be systematically aggregated across layers and views. As a result, developers are often forced into ad hoc implementations rather than guided systematic recombination. To bridge this gap, Libra+ introduces a structured vocabulary that decouples an interaction's trigger, target, and feedback, enabling the systematic description, reuse, and composition of interaction techniques in complex systems.

2.2 The Libra Interaction Model

The Libra model is an architecture for managing and enriching data visualization interactions. Libra's design resembles the reactive principles of modern frontend frameworks such as Vue, extending these structural abstractions from generic UI components to the specialized domain of direct manipulation. As a higher-level framework, Libra+ builds on Libra's architectural abstractions to support more complex and composable interaction patterns. To enable the design, reuse, and combination of interactive elements without modifying underlying charts, Libra formalizes interactions as:

$$\textit{Interaction} := (\text{name, layers, instruments, effects})$$

These represent the interaction's identity, target, means, and results. The *name* uniquely identifies the interaction. The remaining components are described as follows:

Layers: Libra targets distinct, stacked visual planes to prevent interactions from interfering with base visualizations. Each layer creates and updates its elements via a *graphical transformer*, which handles the visual mapping and rendering stages of the pipeline. The model relies on four standard layers: *Background* (static context like axes), *Main* (primary data visualization), *Selection* (isolated visual highlights for selected data), and *Transient* (temporary interactive objects, like a user-drawn bounding box).

Instruments: Instruments mediate between physical inputs and visualization layers, formally defined as:

$$\textit{Instrument} := (\text{name, interactors, services})$$

They process inputs via two sub-components: *Interactors* (state machines translating raw events like `mousedown` into high-level actions) and *Services* (which execute underlying data transformations, such as querying items or running analysis). Services then share their results with graphical transformers to update the corresponding visual layers.

Effects: These are the final outputs, encompassing visual changes and data logging. They provide visual *feedforward* (action previews) and *feedback* (final visual confirmation). Internally, services package data modifications into a *Command* object sent to a global history manager, ensuring every interaction is fully reversible via a robust undo/redo system.

Using the prototype *Libra.js*, developers specify interactions through a declarative JavaScript API. By utilizing simple operators (`inherit`, `insert`, `flow`, and `override`), they seamlessly wrap existing SVG charts with provided instruments and chain data services together. However, this model presents two notable drawbacks. First, an overwhelming number of internal concepts creates a steep learning curve for custom behaviors. The framework lacks a standardized methodology for defining new instruments, leading to highly arbitrary naming conventions. For instance, instruments are frequently named after low-level physical actions, such as `ClickInstrument` or `HoverInstrument`, rather than their semantic purpose. Second, although the framework enables the creation of complex behaviors by combining modular components, effectively coordinating and integrating multiple distinct interactions in practice remains a significant challenge.

3 LIBRA+

In this section, we present the design goals for the Trigger Target Feedback (TTF) model, detail its essential components, and introduce a taxonomy of atomic interactions to facilitate their composition.

3.1 Ambiguity in Interaction Vocabulary

Although interaction is central to data visualization, its foundational vocabulary remains notoriously unstable. Dimara and Perin [10] highlight this severe ambiguity: identical user behaviors often receive disparate names, while single terms frequently describe entirely different system aspects, leaving the field without a precise, consensual vocabulary.

This ambiguity is evident in ubiquitous operations like *brushing* [3], which is routinely used to describe the physical action of dragging a cursor, the computational state of the selected data, and the resulting visual highlighting. When expanded to *brushing and linking* [15], it becomes unclear whether the term denotes an input technique, a coordination mechanism, or a visual effect. Similarly, distinct concepts like *selection*, *highlighting*, and *filtering* are frequently conflated.

Further confusion arises when interface components are collapsed with their underlying semantics. For example, *dynamic queries* [1] (interactive filtering via UI controls) and *scented widgets* [28] (controls augmented with visual feedforward) are often grouped together due to shared UI paradigms. This obscures the critical distinction between query execution and visual guidance.

These terminological overlaps form a fundamental barrier to teaching, system interoperability, and design reuse. While existing taxonomies categorize high-level user intents, they lack the granularity to specify how interactions are physically triggered, which data models they target, and how results are rendered. This gap directly motivates our structured decomposition of interaction into *Trigger*, *Target*, and *Feedback*, providing a precise, mechanistic foundation for specifying and composing interactive techniques.

3.2 The Trigger-Target-Feedback Model

As shown in Fig. 1, an interaction instrument within a visualization view can be formally defined as the following tuple:

$$\textit{Interaction} := (\textit{trigger}, \textit{target}, \textit{feedback}),$$

where *trigger*, *target*, and *feedback* represent the physical user input that initiates the sequence, the specific elements being acted upon, and the system's resulting visual or computational response, respectively. To systematically resolve the ambiguities present in existing interaction vocabularies, these three components are detailed as follows:

- *Trigger:* Bridges the *interactor*, which processes raw low-level events, with high-level semantic intent. For example, canvas panning and rectangular brushing can both be initiated by the same interactor that manages the event sequence `mousedown`, `mousemove`, and `mouseup`. However, they imply entirely different user intents and require distinct feedback mechanisms, as brushing involves drawing a transient rectangle to select specific visual marks.
- *Target:* Defines the scope of the interaction across distinct layers. From an event dispatch perspective, it dictates the specific boundary an event must fall into to trigger a state change within the interactor. Rather than treating the visualization as a monolithic canvas, the target identifies the active domain capable of initiating a state transition by specifying which layer serves this role. For example, in the ordering instrument (see Fig. 1(c)), the *Target* layer comprises the header or axis layers, which the user drags to initiate a change. Targeting specific layers ensures the interactive behavior remains independent of the visual representation while maintaining precise control over entity responsiveness.
- *Feedback:* Determines the visual updates and data transformations executed in response to the interactor's state transitions. It represents the functional realization of the interaction, combining data manipulation via Libra *Services* and graphical rendering through *Transformers*.

While systems like Libra enable customizable feedback through services and transformers, they require coordinating communication across components via shared variables, which form a complex directed network. To address this complexity, we abstract the feedback into four core components for specification:

- *Redraw Function*: Dictates how the visual representation updates dynamically in response to user-triggered events.
- *Interaction Context*: Supplies the essential metadata required for an instrument's execution. This includes coordinate scales, data field names, references to source layers, and specific parameters needed to synchronize any coupled instruments.
- *Transient Feedforward*: Governs the spatial layout (e.g., positional alignment) and stylistic properties (e.g., size, color) of feedback marks. These marks may be generated from scratch or derived directly from source layers containing the primary data marks that respond dynamically to the interaction.
- *Data Service*: Determines the semantic logic of the interaction. It specifies how underlying data structures are reorganized when manipulated objects reach their target destinations. For example, the data service of reorder instrument in Fig. 1 needs to ensure that the relative ordering of adjacent data elements is preserved.

Building upon this model, we introduce parameterized templates that capture common semantic behaviors using predefined services and transformers. By specifying only a small set of parameters to configure the feedback, developers can express interactions through a concise vocabulary rather than low-level implementation details. This parameterization allows feedback behaviors to be specified compactly while maintaining flexibility, ensuring that both visual updates and data transformations are handled consistently. For example, Fig. 1(d) illustrates a template for the reordering instrument. By adjusting just five parameters, this single template can apply reordering behaviors across diverse visualizations, such as matrices, scatterplot matrices (SPLOMs), parallel coordinates, and any categorical axis. Furthermore, as demonstrated in Fig. 1(e, f), ordering instruments configured with different parameters can be combined to create composite interactions.

3.3 Atomic Interaction Instruments

To enable the modular assembly of complex interactions, we distill common patterns into a core set of reusable *Instruments*, sufficient to implement the representative interactions we encountered in the visualization literature. These atomic components capture recurring interaction semantics and can be composed to support advanced behaviors. As summarized in Tab. 1, we group them into four operational categories: *selection*, *rearrangement*, *navigation*, and *visual aids*. The table outlines specific instruments within each category (e.g., *Group Select*, *Move*, *Zoom*, *Lens*), along with their typical physical triggers and corresponding representative feedback options. Each instrument provides customizable default feedback behaviors (such as highlighting, scaling, or invoking specific redraw functions), balancing concise authoring with the flexibility required for diverse visualization scenarios.

Table 1: Taxonomy of Our Atomic Instruments: Categorizing interactions by their Trigger and Feedback options.

Category	Instruments	Triggers	Feedback Options
Selection	Point Select	Hover, Click	Highlight, Tooltip
	Group Select	Brush, BrushX, BrushY	Highlight
	Axis Select	BrushX, BrushY	Highlight
Rearrangement	Move	Drag	Redraw Function
	Reorder	Drag	Redraw Function
Navigation	Pan	Pan	Scale
	Zoom	Zoom	Scale
Visual Aid	Lens	Hover	Radius
	Helper Line	Hover	Direction

- *Selection* instruments identify data via spatial regions or visual properties without altering the underlying visualization structure. Default feedback behaviors include highlighting or dimming marks, filtering subsets, linking selections across views, and showing tooltip-style details for point selections.
- *Rearrangement* instruments modify structural organization, often triggering layout or data-order recomputation (for example, *move* and *reorder*). Defaults include direct graphical manipulation for *move* and matrix-style sorting for *reorder*, while remaining overridable so that chart-specific feedback logic can be supplied.
- *Navigation* instruments adjust the visible data subset or mapping without changing structural composition, such as panning and zooming. They are implemented by updating scale mappings, and authors specify only which scales should be affected.
- *Visual Aids* instruments introduce transient graphical elements that reveal supplementary, otherwise unencoded information derived from data subsets or aggregations. Examples include a *lens* that aggregates data within a spatial neighborhood, or a *helper line* that displays intersecting values on line or area charts for precise contextual reading.

While inspired by Yi et al.'s intent-based taxonomy of seven interaction techniques [30], we consolidate those intents into a smaller, operational set of primitives grounded in system behavior rather than user labels. In particular, *connect*, *filter*, and *explore* can be expressed as *selection*-driven interactions combined with linked layers (see Sec. 4), conditional rendering, or viewport changes; *reconfigure* and *encode* correspond to *rearrangement* and *navigation*; and *abstract/elaborate* is realized via *navigation* or *visual aids*. By factoring these high-level intents into four composable operational primitives, we reduce specification complexity without sacrificing expressive power.

To illustrate how these atomic primitives combine into sophisticated behaviors, Fig. 2 shows the specification of a multi-functional brushing tool on a scatterplot. Instead of implementing a monolithic brush from scratch, developers compose several instruments. First, a base *selection* instrument (Fig. 2a) targets the main data layer to create the brush region and highlight selected marks. Next, a *rearrangement* instrument (Fig. 2b, move) and a *navigation* instrument (Fig. 2c, zoom) are combined with this base instrument. Crucially, these secondary instruments operate on the *transient layer*, the graphical representation of the brush itself, rather than on the underlying data. By mapping drag and zoom triggers to translation and resizing of the brush's bounding box, this example demonstrates how targeting specific layers enables modular yet expressive interaction design within our model.

4 Composition of Interactions

Real-world visualization systems often require multiple instruments to operate concurrently. For example, users may perform selection, navigation, and rearrangement within a single view or across coordinated

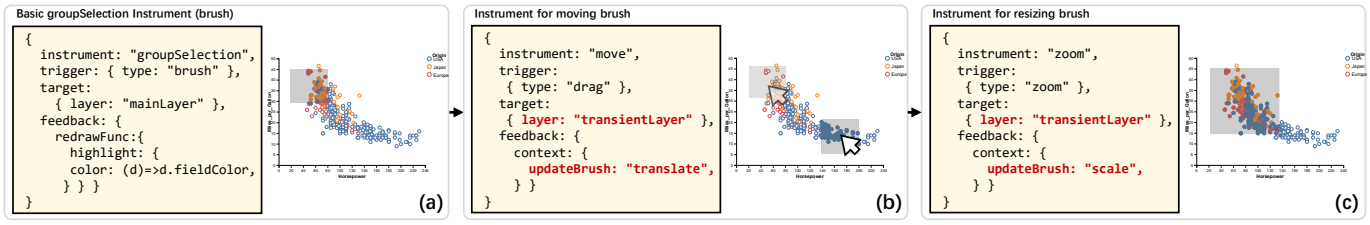


Fig. 2: Specifying a multi-functional brushing interaction on a scatterplot by composing a group selection instrument with move and zoom instruments. A base *group selection* instrument (a) creates the brush, while a *rearrangement* instrument (b, move) and a *navigation* instrument (c, zoom) are applied to the transient layer to enable translation and resizing.

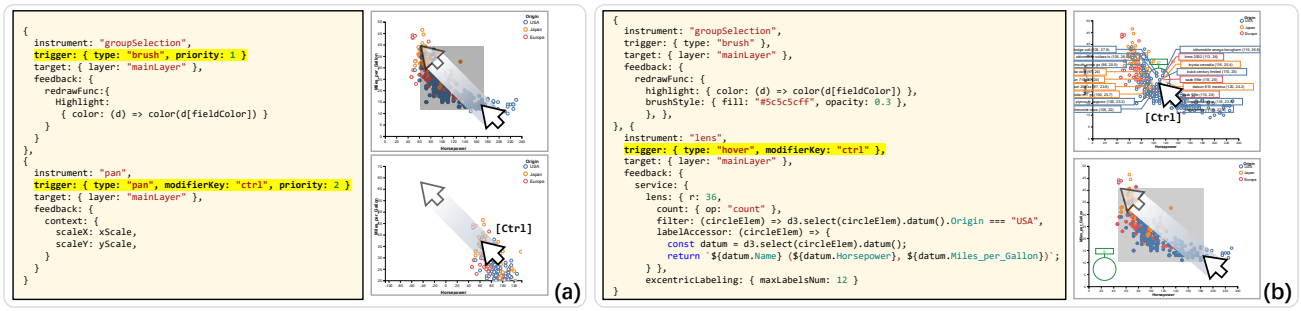


Fig. 3: Examples of resolving interaction conflicts. (a) Using the `priority` and `modifierKey` properties to resolve a trigger and target conflict between brushing and panning. (b) Utilizing modifier separation to resolve feedback occlusion, explicitly tying the lens instrument to the *Ctrl* key so it does not interfere with the active brush.

views. Composing such interactions introduces challenges in managing overlapping inputs, shared visual layers, and potentially conflicting updates. To address these challenges, we formalize relationships between instruments and define coordination mechanisms that ensure consistent and predictable behavior.

4.1 Resolving Conflicts Between Instruments

When multiple instruments are active simultaneously, their interactions depend on input events, layers, and system state. We identify three fundamental relationships for reasoning about interaction composition:

- *Independent:* Instruments operate on disjoint inputs or layers without interference. For example, the base brush instrument (Fig. 2a) and the move/resize instruments (Fig. 2b–c) target different layers (main versus transient), allowing them to coexist without coordination.
- *Exclusive:* Instruments compete for the same input event or control flow, permitting only one to be active at a time. For instance, panning and brushing on the same canvas (Fig. 3a) share the same drag trigger and are therefore mutually exclusive, requiring a mechanism to resolve precedence.
- *Preemptive:* Instruments overlap in visual scope or feedback, requiring explicit ordering to prevent occlusion. For example, the lens and group selection instruments (Fig. 3b) do not conflict in triggers or targets, but their visual feedback may overlap during interaction, requiring coordination to maintain visibility.

Conflict Resolution Strategies. Based on these relationships, Libra+ enables resolving conflicts across triggers, targets, and visual feedback:

- *Trigger Conflicts:* When identical input events occur across multiple layers, the system uses the visual layer hierarchy to route interactions to the intended target (e.g., prioritizing a brush handle over the background).
- *Trigger–Target Conflicts:* When instruments overlap in both input and effect, such as pan and brush mapped to drag on the same layer, developers can apply several strategies to resolve it.
- *Feedback Conflicts:* When visual outputs overlap without trigger conflicts (e.g., lens and brush overlays), the same strategies are applied as for *Trigger–Target Conflicts*.

To resolve these conflicts, users can follow several strategies:

1. *Modifier Separation:* Differentiating actions using keyboard modifiers, for example *Ctrl* for pan and *Shift* for brush.
2. *Priority Sequencing:* Assigning precedence levels so higher priority instruments intercept events.
3. *Pattern Analysis:* Inferring user intent from interaction patterns, such as drag direction, and emitting *synthetic events*.

In Fig. 3b, assigning a modifier key to suppress the lens allows both instruments to operate without visual occlusion. In Fig. 3a, conflicts between pan and brush are resolved through a combination of priority and modifier settings: without a modifier, brushing proceeds normally, while activating a modifier allows the higher-priority pan instrument to intercept the event and suppress brushing. Alternatively, assigning modifiers can eliminate the conflict entirely. For more ambiguous cases, pattern-based synthetic events can be used. In matrix reordering (Fig. 1a), dragging on cells to reorder introduces ambiguity between row and column operations, which can be resolved by using *start vertical* and *start horizontal* events that infer the intended axis from the dominant cursor movement within a short delay (e.g., 500 ms). Similarly, *idle* and *motion* events activate a lens only after the cursor remains stationary for a brief period, avoiding interference during active brushing. These mechanisms take effect by blocking (via modifiers or priorities) or delaying (via pattern analysis) low-level events before the interactors within instruments consume them. Specifically, an instrument may be rendered inactive during event dispatching due to the state of a modifier key, or it may fail to receive a low-level event if a higher-priority instrument has already consumed it.

Combining instruments introduces the issue of conflict and its resolution, which adds to the learning cost. We believe this issue is the price to pay for the ability to combine interactions; we help users detect these conflicts, but their resolution should be handled manually in most cases using `Libra+.js`'s mechanisms. To support this, our online gallery features a tutorial that pairs a reference guide with a hands-on exercise, where users can practice composing instruments and resolving resulting conflicts firsthand.

4.2 Coordinating Instruments Across Layers

Coordinating interactions across different linked views is challenging in existing grammar-based languages and libraries. For example, interaction signals are intermixed with rendering logic in Vega and Vega-Lite, requiring manual maintenance of complex dependency networks to support simultaneous behaviors, such as linked brushing across views. In Libra, components communicate via shared variables, forcing designers to explicitly manage state and rely on shared scopes even when the graphical components are logically distinct. These tight-coupling

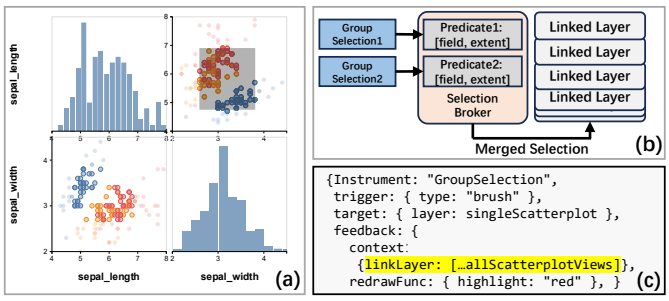


Fig. 4: Coordinating cross-filtering using a publish-subscribe mechanism. (a) Linked brushing applied across a scatterplot matrix. (b) Interaction instruments act as independent publishers, sending selection predicates to a central Selection Broker. The broker merges these states and broadcasts the final selection to all subscribed linked layers. (c) This many-to-many coordination is achieved declaratively by simply passing `[...allScatterplotViews]` to the `linkLayer` property in the instrument's feedback context.

approaches inherently complicate multi-layer and multi-view coordination, often leading to fragile code as visualizations become more complex. Libra+ addresses these limitations by formalizing coordination through a robust, decoupled *Publish–Subscribe* mechanism [11]. This architecture replaces explicit point-to-point bindings with a centralized messaging infrastructure, operating through three distinct roles:

Publisher (Instruments): Instead of directly manipulating visual elements or maintaining hardcoded references to target views, interaction instruments act as independent publishers. They asynchronously broadcast abstract state changes, such as selection predicates, spatial extents, or temporal parameters, to a central message broker. This ensures the instrument remains entirely anonymous and agnostic to its consumers, focusing solely on emitting high-level interaction intents rather than rendering logic.

Broker (Central Dispatcher): Acting as the single source of truth for the interaction state, the broker strictly decouples publishers from subscribers. It receives, stores, and seamlessly merges incoming predicates from multiple simultaneous publishers. Crucially, the broker handles complex state aggregation—such as computing the intersection (AND) or union (OR) of multiple selection boxes drawn across different views before resolving the final state and notifying registered consumers.

Subscriber (Target Layers): Target layers, designated via declarative parameters such as `linkLayer`, act as independent consumers. They attach dedicated modifier components (e.g., `LinkSelectionTransformers`) that dynamically subscribe to the broker. Subscribers selectively listen for relevant state changes to render their specific visual feedback, such as dimming unselected marks or drawing auxiliary guides.

This decoupled architecture introduces three critical characteristics that significantly enhance interaction authoring. First, isolating interaction logic (publishers) from visual feedback (subscribers) enables authors to radically change how a selection is rendered across different views without ever modifying the instrument that triggered it. Second, a single interaction can seamlessly drive updates across dozens of linked views (one-to-many), or multiple interacting instruments can collaboratively filter a single dataset (many-to-one), without an explosive growth in code complexity. Lastly, because communication is anonymous, new views, auxiliary layers, or transient instruments can be dynamically added to the coordination network on the fly. They simply subscribe to the broker without requiring any rewiring of existing components. Ultimately, this mechanism not only facilitates multi-view coordination, as seen in the scatterplot matrix in Fig. 4, but also supports intricate single-view scenarios with multiple linked layers, as illustrated by the DimpVis example in Sec. 6.1.

5 Implementation of Our Model

We implement our model as a prototype `Libra+.js` built on top of Libra.js, centering its abstractions around interaction intent. The specification uses a declarative, data-oriented JavaScript notation that defines a tuple of four components: `instrument`, `trigger`, `target`, and `feedback` (Fig. 1). The `instrument` acts as a handle to select a pre-defined template, while the remaining components instantiate the semantic structure.

Our specification format is not a strict JSON but a JavaScript-based declarative object notation. This design supports flexible customization through callback functions and structured metadata. Developers can customize feedback at three levels: (1) parameter tuning, (2) custom redraw functions, and (3) full pipeline rewiring. The system also provides dispatch controls to resolve conflicts between overlapping instruments and mechanisms, facilitating coordinated updates across multiple layers and views.

5.1 Customizing Feedback

To accommodate substantial variation in interaction feedback across visualizations, Libra+ provides three levels of feedback expressiveness. This design allows authors to rely on parameterized defaults whenever possible, while still supporting more specialized redraw behavior and fully customized Libra.js workflows when needed.

Level 1: Parameterized Feedback. By reusing and composing built-in atomic instruments, authors can customize the appearance of transient layers or the visual properties of the resulting update. This is achieved by specifying a small set of parameters in the parameterized feedback object. For example, in Fig. 2a, using the groupSelection instrument (triggered by brushing) only requires defining the highlight styles. This level handles standard visual adjustments, such as changing a highlight color or adjusting a stroke width. Predefined instruments for selection, navigation, and visual aids can all be used in this declarative manner.

Level 2: Custom Redraw. For certain instruments, such as move and reorder, the underlying intent is consistent across visualizations, but the concrete visual update depends on the chart's structure (Fig. 1). To address this, `Libra+.js` preserves the interaction semantics while delegating rendering to a user-defined `redrawFunc`, paired with a structured `context` that encodes chart-specific metadata such as scales, offsets, and layout parameters.

Beyond custom redraws, `Libra+.js` introduces compiler hints that redirect execution without changing the interaction's conceptual role. For example, `updateBrush` (Fig. 2b-c) serves as a hint within the feedback of move or zoom instruments. When present, the compiler resolves the associated *groupSelection* instrument and forwards the transformation delta to it, enabling brush translation or resizing. Thus, `updateBrush` defines a specialized execution path rather than a new interaction type.

Level 3: Custom Pipeline. For interactions that fall outside the scope of the default feedback mechanisms described in Sec. 3.3 and Tab. 1, authors can define a `customFeedbackFlow`, treating feedback pipeline as a directed graph of Libra.js services and transformers. This enables users to insert, remove, or rewire components within the pipeline while preserving the overall structure. It is particularly useful when multiple triggers share a common but non-standard computation. A representative example is a Dust and Magnet interaction, where different triggers (click and drag) invoke a shared flow to recompute force-directed positions and synchronize the display.

5.2 Our Compiler

A compiler translates high-level specifications into concrete Libra.js constructs, including instruments, services, transformers, and the coordination logic required for composition-aware execution. During the runtime, our prototype extends the Libra.js runtime with compilation, dispatch, coordination, and lookup mechanisms that make concurrent instruments predictable and interoperable.

Addressing Instruments Conflicts. When a low-level event occurs, the compiler performs hit testing to identify the target layer and candidate instruments. In Libra.js, events propagate top-down across layers and stop propagating to underlying layers once an instrument handles the event. Building on this, our prototype introduces a richer dispatch model for complex interaction composition by evaluating candidates within a priority-aware pool rather than relying solely on layer order.

Within the same priority level, layer order and hit testing are preserved, while across levels, higher-priority instruments are considered first, and lower-priority ones are skipped once a handler is found.

Dispatch is further refined by `modifierKey` and `syntheticEvent`: the former separates triggers by keyboard state, and the latter derives higher-level interaction states such as `motion` or `start-horizontally`. They resolve conflicts, such as pan vs. brush or horizontal vs. vertical reordering, without explicit mode switching, enabling a structured and consistent dispatch process.

Note that `Libra+.js` assists in identifying interaction conflicts, particularly those involving overlapping triggers or targets. However, feedback conflicts are difficult to detect automatically and typically require manual inspection. While resolution is ultimately left to the programmer, `Libra+.js` facilitates this process by monitoring potential trigger- or target-related conflicts. If a set of instruments exhibits such conflicts without any declared resolution primitives, the system flags them in an interaction hint panel, displaying the specific triggers (e.g., brush, pan, drag, or click) and targets involved to guide manual resolution.

Coordination across Layers. To enable coordination via a publish–subscribe mechanism, we introduce two concrete implementation forms: a `selectionBroker` for selection instruments and a `genericBroker` for general-purpose component communication.

The `selectionBroker` facilitates selection-based coordination. Selection instruments publish abstract predicates containing a `field` and an `extent`, to which linked layers subscribe via attached transformers. The broker maintains the shared selection state, aggregates incoming updates, and broadcasts them to all subscribers. Linked views typically update in a non-destructive manner by rendering overlays on dedicated selection layers rather than modifying the original marks. This allows each view to independently control its rendering of the shared state without altering the originating instrument.

In contrast, the `genericBroker` enables flexible data exchange with no restrictions on data formats. Its behavior is not limited to standard broadcasting; instead, authors can utilize these brokers within custom feedback flows to actively pull and parse data within specific services or transformers. For example, in a DimpVis-like interaction, a single service may publish interpolated intermediate states while multiple downstream transformers consume these states to update related nodes and labels simultaneously. Thus, the broker serves as a versatile coordination substrate, supporting both standard linked selections and complex cross-component communication.

Context-aware Layers. While the publish–subscribe mechanism enables efficient cross-layer communication, it can be computationally expensive and architecturally redundant for resolving immediate interaction contexts across dynamically generated layers. Instead of relying on a global broker for every transient update, our compiler treats layers as structured entities within a hierarchical tree.

When an instrument creates auxiliary layers such as selection or transient layers, Libra+ explicitly records parent–child relationships. This hierarchy allows nested instruments to dynamically inherit the necessary spatial context. For example, in Fig. 2b-c, the selection range of a `groupSelection` instrument can be modified by secondary *move* or *zoom* instruments. Rather than querying a central broker, these instruments traverse the hierarchy to the parent layer, directly accessing the original coordinate scales and brush extents.

Furthermore, this hierarchy ensures automated synchronization. For selection instruments, once a target main layer is redrawn or its data transformed, the associated selection results must update synchronously. By registering lifecycle listeners within the layer tree, Libra+ triggers an automatic refresh of all child selection layers whenever the parent layer changes. This approach maintains visual and logical consistency without the overhead of manual state synchronization.

6 Evaluation

To demonstrate reusability and composability, we evaluate Libra+ along three technical axes: its ability to reproduce sophisticated techniques from prior work, generalize interaction templates across diverse visualization contexts, and enrich interactions by combining and extending multiple instruments.

6.1 Reproduce

We reproduce sophisticated techniques from prior work (i.e., cross-filtering and DimpVis) within a unified architecture to demonstrate its capacity to express complex, multi-component behaviors and custom feedback consistently.

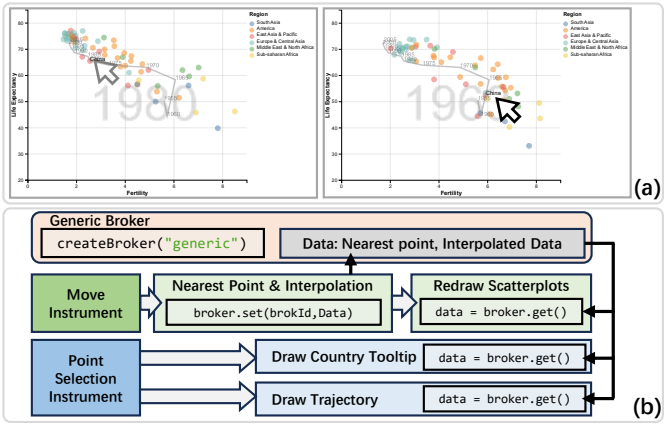


Fig. 5: Constructing the DimpVis interaction in `Libra+.js`. (a) Direct manipulation of a data point's trajectory triggers global temporal updates. (b) The underlying interaction composition resolving cross-instrument coordination. The custom feedback flows of the `MoveInstrument` and `PointSelectionInstrument` interact seamlessly via a shared generic broker. (Block arrows: control flow; Solid black arrows: data flow)

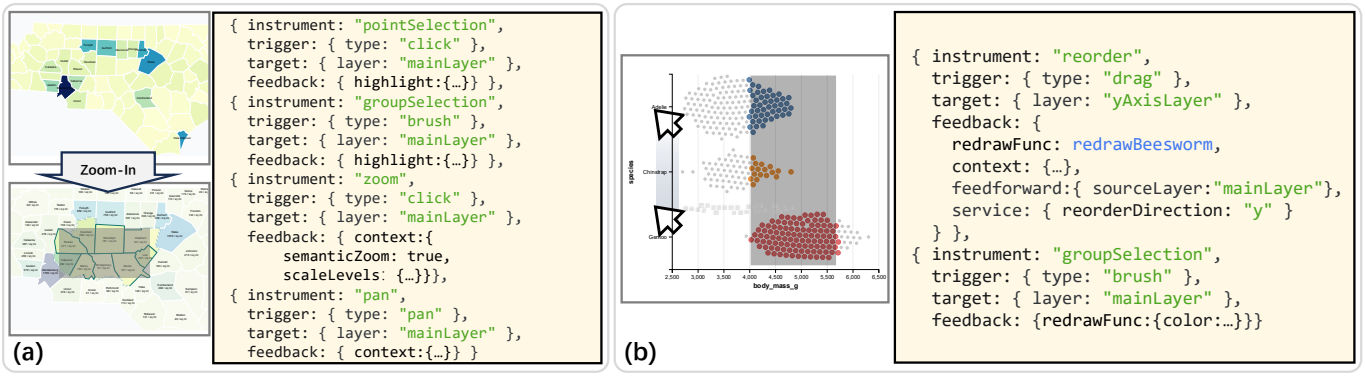
Cross-Filtering. As shown in Fig. 2a, multi-view cross-filtering allows users to brush a region in one scatterplot and instantly highlight related data across all linked views. Managing decentralized interaction states is the core challenge here. Traditionally, hardcoding these multi-view dependencies creates tightly coupled, monolithic components.

`Libra+.js` overcomes this by natively managing synchronization through its broker architecture and declarative context linking. We simply deploy a `GroupSelection` instrument with a `brush` trigger. During interaction, as illustrated in Fig. 2b, this instrument extracts predicates defining the spatial extent and publishes them to a central `SelectionBroker`. The broker then merges these predicates and broadcasts the unified state to all target layers. Crucially, as detailed in Fig. 2c, this global coordination is achieved declaratively by simply passing the target views into the `linkLayer` context array.

DimpVis. As shown in Fig. 5, DimpVis [17] couples direct manipulation with complex, coordinated downstream updates. Hovering over a data point reveals its country label and draws its trajectory across all time steps. Dragging the point along this line derives the interpolated time from the cursor position, simultaneously moving all other points to their corresponding temporal coordinates.

Instead of relying on rigid custom code, Libra+ reconstructs DimpVis by composing two atomic instruments: a `PointSelectionInstrument` and a `MoveInstrument`. Because this behavior exceeds simple parameter adjustments, both utilize Level-3 custom feedback flows. The selection instrument manages trajectory filtering and label rendering, while the move instrument calculates the nearest trajectory point, computes the temporal interpolation, and updates the main view. Operating on orthogonal triggers, these atomic units compose seamlessly without input conflicts.

The primary technical challenge in this reproduction lies in the coordination between instruments. The country label must dynamically follow the selected point as it moves. However, because the `PointSelectionInstrument` renders this label, dragging the cursor away from the original target breaks the hover trigger, causing the system to lose the natively captured position. Libra+ resolves this issue through its native broker architecture. As illustrated in Fig. 5b, the drag flow's `InterpolationService` computes the updated coordinates




```
{ instrument: "pointSelection",
  trigger: { type: "click" },
  target: { layer: "mainLayer" },
  feedback: { highlight:{…}} },
{ instrument: "groupSelection",
  trigger: { type: "brush" },
  target: { layer: "mainLayer" },
  feedback: { highlight:{…}} },
{ instrument: "zoom",
  trigger: { type: "click" },
  target: { layer: "mainLayer" },
  feedback: { context:{
      semanticZoom: true,
      scaleLevels: {…}}},
{ instrument: "pan",
  trigger: { type: "pan" },
  target: { layer: "mainLayer" },
  feedback: { context:{…}} }
```

```
{ instrument: "reorder",
  trigger: { type: "drag" },
  target: { layer: "yAxisLayer" },
  feedback: {
    redrawFunc: redrawBeesworm,
    context: {…},
    feedforward:{ sourceLayer:"mainLayer"},
    service: { reorderDirection: "y" }
  } },
{ instrument: "groupSelection",
  trigger: { type: "brush" },
  target: { layer: "mainLayer" },
  feedback: {redrawFunc:{color:…}}}
```

Fig. 6: Generalizing Libra+ instruments across visualization contexts: (a) a choropleth map demonstrating the composition of semantic zooming, panning, and selection; (b) a beeswarm plot illustrating custom axis reordering through drag interactions and group selection via brushing.

and publishes this state to a `genericBroker`. The `textTransformer` then seamlessly pulls this data to accurately position the label. Ultimately, this example demonstrates that even highly customized interactions can be deconstructed into atomic components and orchestrated flawlessly within Libra+'s unified compositional framework, eliminating the need for monolithic, black-box designs.

6.2 Generalize

We evaluate whether Libra+ instruments generalize beyond the standard chart types. The key question is whether composed interactions can be consistently applied across substantially different visual representations. Fig. 6 illustrates this capability through two representative cases.

Choropleth Map. Fig. 6a demonstrates consistent composition and configuration of interactions of a choropleth map with a semantic hierarchy. Users can brush or click to highlight regions and navigate through panning and semantic zooming. This behavior is realized by composing four atomic instruments: `GroupSelection`, `PointSelection`, `Pan`, and `Zoom`. All are deployed through parameterized templates. `GroupSelection` specifies highlight styles, while `Pan` and `Zoom` operate on the X and Y scales. The `ZoomInstrument` also encodes semantic zoom rules and hierarchical data in its `context`.

Conflicts between brushing and panning are handled through built-in coordination mechanisms. Assigning a `modifierKey` (e.g., Shift) separates navigation from selection, while updating the main layer automatically redraws the layers associated with the `GroupSelection` instrument. This ensures that the `selectionLayer` remains consistent under viewport changes.

Beeswarm Plot. Fig. 6b shows a beeswarm plot with a custom categorical axis that supports both row reordering and brushing. This is achieved by composing `GroupSelection` and `Reorder` instruments, illustrating how Libra+ supports multi-functional interactions through modular composition rather than bespoke implementations.

The selection instrument requires only declarative configuration of its `target.layer` and highlight behavior. The `ReorderInstrument` retains the same structure as in Fig. 1, and generalization is achieved by supplying a chart-specific `redrawFunc` and passing the target list and scale through `feedback.context`. Crucially, this setup benefits from native, automatic selection re-evaluation due to the design of context-aware layers. When reordering triggers a global redraw, the brush region and highlights update automatically, maintaining consistent synchronization without additional coordination mechanisms.

6.3 Enrich

We demonstrate how Libra+ simplifies the enhancement of existing applications. Rather than rebuilding a tool from scratch to add a new feature, `Libra+.js` allows developers to additively compose new instruments onto an established design: they can simply "stacking" new instruments within the same system.

Extended Dust-and-Magnet. Fig. 7 illustrates an extension of the Dust-and-Magnet [31] technique that incorporates excentric labeling [12]. The base application maps multiple interaction triggers to a

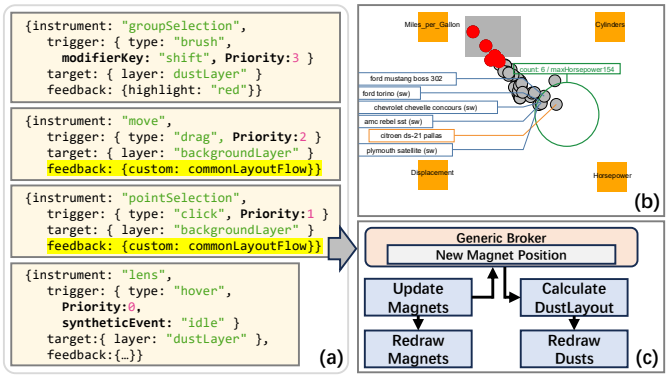


Fig. 7: Implementing an enhanced Dust-and-Magnet visualization with excentric labeling in Libra+ by composing multiple distinct instruments into a unified custom feedback flow via centralized broker coordination.

unified feedback flow. Specifically, clicks to create magnets and drags to reposition them invoke a single shared sequence to update positions, recompute the dust layout, and refresh all affected marks.

To enrich this visualization, we add an excentric labeling instrument to resolve severe visual clutter. As magnets attract the data points, the dust frequently forms highly dense, overlapping clusters. By additively composing this labeling instrument onto the chart, users can hover over these congested areas to extract local details instantly. The instrument captures the underlying points and dynamically projects their textual labels outward into a readable, non-overlapping circular layout.

This addition demonstrates structural enrichment. The labeling instrument operates independently by retrieving cursor coordinates and dust node positions directly from the dust layer. It computes and renders the peripheral labels without interfering with the complex physics computations of the underlying magnets. Developers achieve this by simply composing the new instrument onto the existing design, proving that Libra+ can seamlessly stack new interactive behaviors to clarify dense visualizations.

Extended Neighboring Edge Highlighting. Fig. 8 illustrates an enriched neighboring edge highlighting technique designed to enhance local graph readability. The base application maps a simple hover interaction to a direct feedback flow, triggering a standard `pointSelection` instrument to highlight all connected edges.

While standard highlighting reveals initial connections, clustered edges frequently obscure one another. For example, in Fig. 8b, highlighting all edges attached to Node 49 fails to clarify distinct connections to Nodes 56, 71, 72, or 42. To enrich this visualization and resolve severe visual clutter, we add an edge lens instrument. By composing a second `pointSelection` instrument [29], users can hold the Shift key to activate this lens. The instrument dynamically distorts the layout; edges near the pointer bend outward while distant edges remain straight, creating a visual channel for local separation. Fig. 8c demonstrates the result of this interaction, clearly revealing that Nodes 56 and 72

connect to Node 49, whereas Nodes 71 and 42 do not.

Both `pointSelection` instruments share identical triggers and targets but operate independently without cross-communication. While the standard `pointSelection` defines its feedback via strict parameterization, the supplementary `pointSelection` instrument implementing the EdgeLens operates at Level 3. It utilizes a `CustomFeedbackFlow` paired with a Shift modifier key. Developers achieve this by replacing the default selection service with an `edgelensLayoutService` and `linkTransformer` chain, which computes the displaced edge geometries prior to rendering. Finally, assigning a modifier key ensures users can effortlessly revert to the original graph layout.

6.4 Comparison of Interaction Authoring

We compare the authoring effort required by `Libra+.js` with that of Libra.js, Vega-Lite, and D3 through side-by-side examples from our online gallery.

Using brushing as a representative case, we find that all frameworks can implement basic single brushing with relative ease, but clear differences emerge as interaction complexity increases. A common extension is multi-brushing, which allows users to create multiple arbitrary brush regions within the same chart. In `Libra+.js`, this can be achieved by either adding a `remnantKey` attribute to the `groupSelection` instrument or composing multiple instances of the instrument. By contrast, implementing the same behavior in Libra, Vega-Lite, or D3 is considerably more cumbersome. Vega-Lite requires low-level event-handling specifications to distinguish concurrent events, and its default "select-all" state can lead to incorrect behavior when interactions are extended. D3 and Libra, meanwhile, require developers to manually manage shared variables to avoid state conflicts.

The gap becomes even more pronounced when brushing is combined with sophisticated interactions such as Dust & Magnet. For such highly customized composite logic, Vega-Lite reaches its architectural limits and cannot express the full interaction. Although D3 and Libra remain capable of implementing these combinations, they incur increasing overhead in data transmission and conflict resolution. In `Libra+.js`, by contrast, code growth is mainly devoted to defining the desired custom feedback rather than managing interaction plumbing. Moreover, the declarative primitives of `Libra+.js` improve learnability compared with the imperative, API-heavy style of D3 and Libra, enabling authors to specify complex behaviors more clearly and concisely.

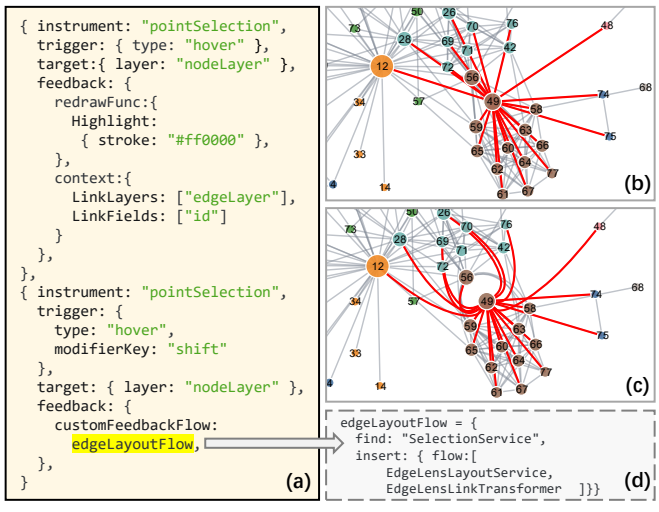


Fig. 8: Structural enrichment of neighboring edge highlighting. The system seamlessly stacks a `EdgeLens` instrument (a) onto the `pointSelection` instrument. Utilizing a custom feedback flow (d), the instrument bends overlapping edges (c) upon a Shift key press, cleanly resolving the visual clutter typical of standard highlighting (b).

7 Discussion and Conclusion

We introduce an interaction model that builds complex interactions for data visualization by composing simple primitive ones. Building on and extending prior interaction architectures such as Libra, our model adopts a concise yet extensible trigger–target–feedback representation that unifies a wide range of existing interactions. We implement this model as a domain-specific language, `Libra+.js`, which includes strategies to resolve conflicts among combined interaction instruments and coordinate their behaviors. We demonstrate that `Libra+.js` is expressive enough to reproduce various interactions, generalizes to custom visualizations, and can be readily extended to support new interactions, although it currently targets 2D visualizations and does not yet support 3D interactions like VisLink [9].

Compared with Libra, Libra+ introduces three main improvements. First, it simplifies instruments, making them easier to extend, compose, and reuse. For example, in the brushing interaction shown in Fig. 2, adding move or zoom behaviors in Libra requires extra instruments, restructuring internal services and transformers, and manually managing shared variables, often resulting in monolithic instruments. In contrast, Libra+ supports these extensions directly at the DSL level without modifying the `groupSelection` instrument. Second, Libra+ provides explicit mechanisms for conflict resolution and interaction coordination. Whereas Libra requires manual coordination for behaviors such as brushing with canvas panning or linked selection, Libra+ expresses these patterns through concise declarative primitives. Finally, Libra+ offers a more declarative specification model while retaining comparable feedback expressiveness through `customFeedbackFlow`, which allows users to define services and transformers via the Libra.js API. For instance, although Libra+ lacks a native lasso trigger, users can implement lasso selection by combining a dragging-triggered move instrument with `customFeedbackFlow`. This enables users to construct novel interactions by composing and adapting a small set of atomic instruments.

Currently, `Libra+.js` supports only static visualizations implemented with D3.js. However, following the extensible design of Libra.js, it could be extended in the future to support other visualization grammars such as Vega and Observable Plot. Nevertheless, Libra+ and `Libra+.js` still have several limitations.

Unstable Vocabulary. Libra+ divides complex interactions into individual atomic instruments; while this approach enhances modularity and readability, it inherently introduces additional descriptive overhead. Furthermore, the space of data interaction is rapidly evolving, with new input modalities and visualization paradigms continuously reshaping what constitutes fundamental concepts such as "trigger" and "feedback." Although Libra+ is extensible, allowing for custom instruments, the lack of a shared ontology in the visualization community can lead to fragmented terminology. Continued refinement is needed to keep the vocabulary expressive without becoming overly complex.

Learning Cost. Specifying interactions in `Libra+.js` is not yet entirely straightforward. To support a wide range of feedback, we include callback functions in the library, which increases expressiveness but also steepens the learning curve. As a result, users with limited programming experience may find authoring Level-2 or Level-3 feedback time-consuming. We will conduct a user study to assess the usability and further refine, reducing manual effort while retaining flexibility.

Performance Optimization. While our current implementation demonstrates the model's capabilities, it has not yet been heavily optimized for performance. Consequently, bottlenecks can arise with specific combinations of visualizations and instruments. For instance, the *reorder* instrument automatically creates a copy of the main visualization layer and redraws it continuously during a drag operation. As a result, users may experience noticeable latency when applying this interaction to visualizations containing a large number of marks.

Intermediate Representation for AI Automation. Although LLM-driven visualization generation [6] is advancing rapidly, the automated generation of sophisticated interactions remains limited by the lack of a consistent interaction vocabulary. Libra+ helps bridge this gap by providing a formal descriptive model for specifying interactions. As future work, we envision using Libra+ as an intermediate representation that enables natural language inputs to be translated by LLMs into `Libra+.js`, which users can then edit or directly compile into interactive visualizations.

Acknowledgment

This work is supported by the grants of the NSFC (No.U2436209), the Beijing Natural Science Foundation (L247027), the Fundamental Research Funds for the Central Universities, andthe Research Funds of Renmin University of China.

References

[1] C. Ahlberg and B. Shneiderman. Visual information seeking: Tight coupling of dynamic query filters with starfield displays. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 313–317, 1994. doi: 10.1145/191666.191775 3

[2] M. Beaudouin-Lafon. Instrumental interaction: an interaction model for designing post-WIMP user interfaces. In *Proc. SIGCHI Conf. Human Factors Comp. Sys.*, pp. 446–453, 2000. doi: 10.1145/332040.332473 2

[3] R. A. Becker and W. S. Cleveland. Brushing scatterplots. *Technometrics*, 29(2):127–142, 1987. doi: 10.1080/00401706.1987.10488204 3

[4] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE Trans. Vis. Comput. Graphics*, 15(6):1121–1128, 2009. doi: 10.1109/tvcg.2009.174 2

[5] M. Bostock, V. Ogievetsky, and J. Heer. D³ Data-Driven Documents. *IEEE Trans. Vis. Comput. Graphics*, 17(12):2301–2309, Dec. 2011. doi: 10.1109/TVCG.2011.185 2

[6] M. Brossier, T. Isenberg, K. Schönborn, J. Unger, M. Romero, J. Björklund et al. State of the art of llm-enabled interaction with visualization, 2026. doi: 10.48550/arXiv.2601.14943 9

[7] S. K. Card, J. D. Mackinlay, and B. Shneiderman. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, 1999. doi: 10.5555/300679 2

[8] J. Choi, D. G. Park, Y. L. Wong, E. Fisher, and N. Elmqvist. Visdock: A toolkit for cross-cutting interactions in visualization. *IEEE Trans. Vis. Comput. Graphics*, 21(9):1087–1100, 2015. doi: 10.1109/tvcg.2015.2414454 2

[9] C. Collins and S. Carpendale. Vislink: Revealing relationships amongst visualizations. *IEEE Trans. Vis. Comput. Graph.*, 13(6):1192–1199, 2007. doi: 10.1109/TVCG.2007.70521 9

[10] E. Dimara and C. Perin. What is interaction for data visualization? *IEEE Trans. Vis. Comput. Graphics*, 26(1):119–129, 2019. doi: 10.1109/tvcg.2019.2934283 3

[11] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131, 2003. doi: 10.1145/857076.857078 6

[12] J.-D. Fekete and C. Plaisant. Excentric labeling: Dynamic neighborhood labeling for data visualization. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pp. 512–519, 1999. doi: 10.1145/302979.303148 8

[13] J. A. Guerra-Gomez. Towards reusable and reactive widgets for information visualization research and dissemination. In *2024 IEEE Visualization and Visual Analytics (VIS)*, pp. 316–320. IEEE, 2024. 2

[14] J. Heer, S. K. Card, and J. A. Landay. Prefuse: a toolkit for interactive information visualization. In *Proc. SIGCHI Conf. Human Factors Comp. Sys.*, pp. 421–430, 2005. doi: 10.1145/1054972.1055031 2

[15] J. Heer and B. Shneiderman. Interactive dynamics for visual analysis: A taxonomy of tools that support the fluent and flexible use of visualizations. *Queue*, 10(2):30–55, 2012. doi: 10.1145/2133806.2133821 3

[16] J. Hoffswell, A. Satyanarayan, and J. Heer. Visual debugging techniques for reactive data visualization. *Computer Graphics Forum*, 35(3):271–280, 2016. doi: 10.5555/3071534.3071564 2

[17] B. Kondo and C. Collins. Dimpvis: Exploring time-varying information visualizations by direct manipulation. *IEEE Trans. Vis. Comput. Graphics*, 20(12):2003–2012, 2014. doi: 10.1109/TVCG.2014.2346250 7

[18] B. A. Myers. Separating application code from toolkits: eliminating the spaghetti of call-backs. In J. R. Rhyne, ed., *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology*, pp. 211–220, 1991. doi: 10.1145/120782.120805 2

[19] T. M. H. Reenskaug. The original MVC reports, 1979. 2

[20] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-lite: A grammar of interactive graphics. *IEEE Trans. Vis. Comput. Graphics*, 23(1):341–350, 2017. doi: 10.1109/tvcg.2016.2599030 1, 2

[21] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE Trans. Vis. Comput. Graphics*, 22(1):659–668, Jan. 2016. doi: 10.1109/TVCG.2015.2467091 2

[22] A. Satyanarayan, K. Wongsuphasawat, and J. Heer. Declarative interaction design for data visualization. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, pp. 669–678, 2014. doi: 10.1145/2642918.2647360 2

[23] Shneiderman. Direct manipulation: A step beyond programming languages. *Computer*, 16(8):57–69, 1983. doi: 10.1109/MC.1983.1654471 2

[24] L. S. Snyder and J. Heer. DIVI: dynamically interactive visualization. *IEEE Trans. Vis. Comput. Graph.*, 30(1):403–413, 2024. doi: 10.1109/TVCG.2023.3327172 2

[25] Z. Wan, W. Taha, and P. Hudak. Event-driven FRP. In *International Symposium on Practical Aspects of Declarative Languages*, pp. 155–172. Springer, 2002. doi: 10.5555/645772.667941 2

[26] C. Weaver. Building Highly-Coordinated Visualizations in Improvise. In *IEEE Symposium on Information Visualization*, pp. 159–166. IEEE, 2004. doi: 10.1109/INFVIS.2004.12 2

[27] L. Wilkinson. The grammar of graphics. In *Handbook of computational statistics*, pp. 375–414. Springer, 2012. doi: 10.5555/316575 1, 2

[28] W. Willett, J. Heer, and M. Agrawala. Scented widgets: Improving navigation cues with embedded visualizations. *IEEE transactions on visualization and computer graphics*, 13(6):1129–1136, 2007. doi: 10.1109/TVCG.2007.70589 3

[29] N. Wong, S. Carpendale, and S. Greenberg. Edgelens: An interactive method for managing edge congestion in graphs. In *IEEE Symposium on Information Visualization 2003 (IEEE Cat. No. 03TH8714)*, pp. 51–58. IEEE, 2003. doi: 10.1109/INFVIS.2003.1249008 8

[30] J. S. Yi, Y. ah Kang, J. Stasko, and J. A. Jacko. Toward a deeper understanding of the role of interaction in information visualization. *IEEE Trans. Vis. Comput. Graphics*, 13(6):1224–1231, 2007. doi: 10.1109/tvcg.2007.70515 3, 4

[31] J. S. Yi, R. Melton, J. T. Stasko, and J. A. Jacko. Dust & Magnet: multivariate information visualization using a magnet metaphor. *Information Visualization*, 4(3):239–256, 2005. doi: 10.1057/palgrave.ivs.9500099 8

[32] Y. Zhao, Y. Wang, X. Luo, Y. Wang, and J.-D. Fekete. Libra: An interaction model for data visualization. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, pp. 1–17, 2025. doi: 10.1145/3706598.3713769 1, 2