Declarative Dragging

Anonymous Author(s)

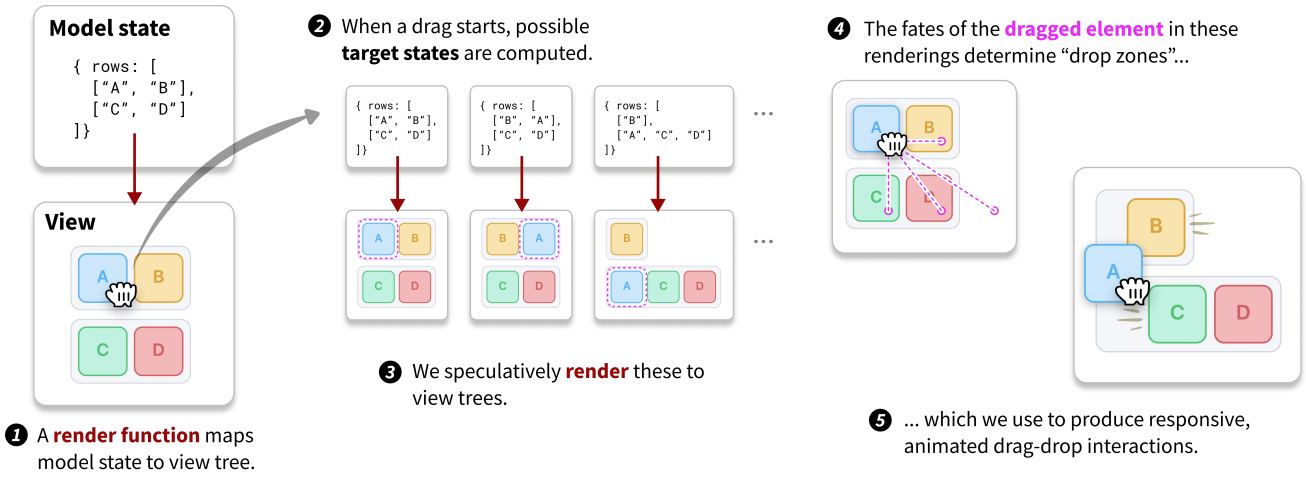

Figure 1: Dragology is a library implementing "declarative dragging": a new way for interface developers to make higher-quality drag-and-drop interactions with less work. Dragology enables *model-driven dragging*. An interface developer writes a render function which turns a model state into a view tree (❶). They also specify, for each draggable element, a set of target model states (❷). When a drag starts, Dragology speculatively renders these target states to view trees (❸). By extracting the location of the dragged element in these speculative renderings (❹), Dragology can synthesize a responsive drag-and-drop interaction featuring high-quality transition animations (❺) – all from just the developer-provided list of model states.

Abstract

Drag-and-drop interactions realize the vision of direct manipulation, allowing objects to be manipulated in intuitive and responsive ways. However, implementing such interactions typically requires managing complex spatial, temporal, and visual behaviors. We present a higher-level alternative, *declarative dragging* (D2). D2 is *model-driven*: An interface author makes an element draggable by specifying which future model states should be accessible by dragging it. An engine speculatively renders these states to view trees, extracts the position of the dragged element in each of these renderings, and uses these to synthesize a drag-and-drop interaction, all without detailed work required by the interface author. To provide greater expressive range and customizability, D2 is *composable*: simple model-driven drag-and-drop behaviors can be combined to create more complex ones, and modifiers can layer features like floating, snapping, and transitions. To test D2, we built a prototype engine called Dragology. We demonstrate that Dragology's model-driven primitives can concisely express a broad range of high-quality drag-and-drop interactions. Interviews with 13 interface developers suggest declarative dragging lowers the floor for building polished, varied drag-and-drop interactions – while surfacing barriers to learning and adoption.

CCS Concepts

• **Human-centered computing** → **Interaction techniques**; **User interface toolkits**; *User studies*.

Keywords

drag-and-drop, direct manipulation, declarative interfaces, interaction design, speculative rendering

ACM Reference Format:

Anonymous Author(s). 2026. Declarative Dragging. In *Proceedings of Conference Name (Conference 'XX)*. ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Direct manipulation (DM) is an interaction paradigm which maximizes *directness* – the "qualitative feeling that one is directly engaged with... the semantic objects of our goals and intentions" [17]. Direct manipulation is exemplified by *drag-and-drop*, which builds analogically from the familiar activity of picking up, moving, and dropping physical objects. Drag-and-drop lets users apply tools and manipulate structures easily and effectively.

However, high-quality drag-and-drop interfaces are hard to build. They require complex spatial logic (*e.g.* computing drop zones),

Conference 'XX, City, Country

ACM ISBN 978-1-4503-XXXX-X/2026/00
https://doi.org/XXXXXXX.XXXXXXX

temporal logic (*e.g.* managing animated transitions), and rendering logic (*e.g.* maintaining previews during drag). A developer can avoid much of this complexity using higher-level APIs or libraries, such as the HTML5 Drag and Drop API [21]. But these limited platforms can only produce formulaic behaviors that lack the continuity and immediacy that make drag-and-drop so appealing. In many cases, the complexity of drag-and-drop causes interface developers to forsake it entirely in favor of simpler forms and widgets.

Our goal is to make it easier for developers to implement high-quality drag-and-drop interactions. We contribute *declarative dragging* (D2), an approach made of two novel ingredients. The first is *model-driven dragging*, where an interface author specifies a drag interaction in terms of model states they wish to make accessible through the drag, without needing to explicitly construct drop zones for each model state. We make this possible through speculative rendering: we render the interface in each possible future state and extract the location of the dragged element from each rendering. The second ingredient is *composable drag behaviors*: constructing complex drag-and-drop behaviors using an expressive language of primitives, combinators, and modifiers that define the visual and functional behavior of interactions.

We implemented D2 in a library called Dragology, built for SVG interfaces on the Web platform.[1] Through demonstrations, we show that Dragology has a broad range of applications where it can express complex drag-and-drop interactions concisely and naturally. To further evaluate D2, we interviewed 13 developers with prior drag-and-drop experience, giving each hands-on experience with Dragology. Participants found that using Dragology resulted in concise code, polished interactions, and easy experimentation. Our study also identified points of friction around state-space construction and reasoning.

In summary, this paper contributes:

(1) *Declarative dragging*, a new approach to building drag-and-drop interfaces combining two novel ingredients: *model-driven dragging* and *composable drag behaviors*.
(2) *Dragology*, an implementation of the declarative dragging approach as a library on the Web platform,
(3) demonstration of Dragology through a large gallery of both familiar and innovative examples, and
(4) a hands-on interview study suggesting D2 can lower barriers to building polished interactions while enabling exploratory design.

2 Background & Related Work

2.1 Implementing Drag-and-Drop Interfaces

Developers building drag-and-drop interfaces typically take one of three approaches. None of these is simultaneously easy to use, expressive, and capable of creating high-quality interactions.

The first is *low-level event handling*, where an interface author attaches handlers to events like "pointer down", "pointer move" and "pointer up". This approach gives a developer full control and maximal expressive range. However, it requires significant work, even for simple interactions – from coordinate calculation to managing the state machine of transient states. Furthermore, this approach makes drag-and-drop a cross-cutting concern, threaded across many parts of an application's code, making modification difficult. While this approach presents no limitations in expressivity and quality, it has high viscosity [14] and requires significant specialized labor.

The second approach is *drag sources and drop targets*, in which certain interface elements are marked as draggable and others are marked as targets for drags.[2] The platform or library handles the rest, such as updating a view of the object as it is dragged. Once an interface is represented in terms of sources and targets, this approach is easier than handling raw events. However this approach is only a small step up in quality from widget-based interaction. Interfaces made with this approach take on a flat appearance, lacking the continuity and domain-specific feedback that make drag-and-drop so effective. Furthermore, the initial ease of this approach often breaks down; for instance, when interstitial elements must be created to serve as drop targets.

The third approach is *specialized libraries*, designed to accommodate particular patterns like reorderable lists. These can be easy to use, and fine-tuned to produce high-quality interactions, but they are inherently limited in the types of interfaces they can support. Developers exploring idiosyncratic interactions will hit boundaries of expressiveness.

Declarative dragging (D2) achieves a unique combination of ease of use, expressiveness, and interaction quality. Across a specific but interestingly diverse range of use-cases, D2 can create appealing interactions with less effort than any of these approaches.

2.2 Functional Views

Model-driven dragging builds on the foundation of *functional views*: defining interfaces as functions from model states to views. While this approach has deep roots (including fudgets [6], functional reactive programming [10], and immediate-mode GUIs [1]), it has more recently risen to prominence through web frameworks like React [20] and Elm [12]. These frameworks offer a more declarative approach than the former status quo, letting developers specify the semantics of how a view is determined by model state while leaving the details of accurate and performant interface updates to the framework. D2 extends this functional approach. While functional-view frameworks let developers define how individual model states map onto individual views, model-driven dragging lets developers define how *transitions* between states should map onto *drag interactions* between views.

Model-driven dragging is based on the insight that existing functional-view systems do not make as much use of their render function as they could, only rendering the model state of an application through sequential updates. Model-driven dragging instead speculatively renders branching future states in order to infer drop zones. Several projects, including Apparatus [32] and g9 [37], use speculative rendering to drive continuous drag-and-drop interactions on diagrams. D2, as embodied in Dragology, extends these to support both discrete and continuous drag-and-drop, using a novel DSL of composable behaviors.

[1] Dragology is open-source, available for use at [GITHUB URL ANONYMIZED FOR SUBMISSION] or on NPM at [PACKAGE NAME ANONYMIZED FOR SUBMISSION]. Interactive figures accompanying this paper are available at https://declarative-dragging.github.io/.

[2] This approach is exemplified by the HTML5 Drag and Drop API [21], as well as libraries like "dnd kit" [8].

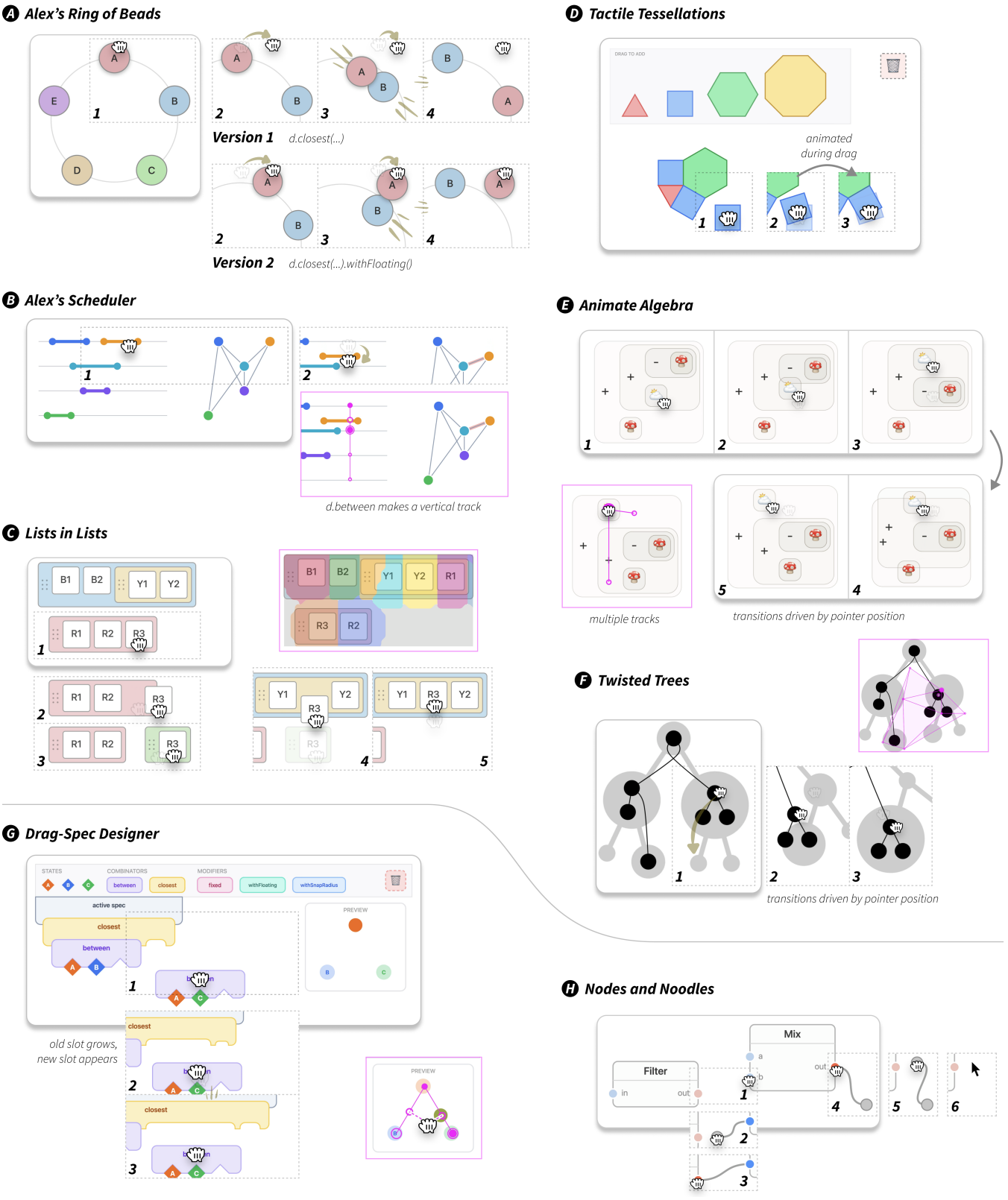

Figure 2: Gallery of interfaces built with Dragology. Ⓐ-Ⓑ are discussed in §4, Ⓒ-Ⓕ in §6, and Ⓖ-Ⓗ in Appendix B.

2.3 Declarative & Composable User Interfaces

D2 fits into a history of efforts to model and implement user interfaces at a high level. "Model-based user interface design" [34] was an ambitious set of efforts to generate user interfaces in their entirety from models of tasks. However, "the UI semantics [of MBUID] were limited to simple components like buttons, sliders and text boxes" [2]. State diagrams and statecharts [15] (and systems built on these like InterState [24]) reify aspects of state that are typically buried in procedural code. D2 offers higher-level abstractions than these; a developer using D2 does not manage transient interaction state by hand. Garnet [22] introduced a notion of "interactors" [23] that encapsulate interaction patterns, which was followed by Vega [31] and Vega-Lite [30]. While some of these projects use composable FRP-inspired signals, they do not make interactions themselves composable, nor do they attain our expressive range.

3 Declarative Dragging

Declarative dragging combines two novel contributions: *model-driven dragging* and *composable dragging*.

3.1 Model-Driven Dragging

Many drag-and-drop interactions share the property that dragging an element leaves it where you drop it: either precisely, in the case of a canvas-style interaction, or nearby, like snapping into a list position. We call such drags *repositional*.[3]

The first insight of D2 is that adding a repositional drag interaction to a functional view ought to require little more than specifying which future model states should be reachable through the drag. Take the example of a Scrabble rack, where the model state is an array of letters like $[A, B, C]$ and a view function maps this to a row of tiles. We want to add a drag-and-drop interaction for rearranging letters on the rack. Dragging a given letter will allow us to *access* certain model states on drop. For instance, if B is dragged, three arrays should be accessible: $[B, A, C]$, $[A, B, C]$ (the starting state), or $[A, C, B]$. In status-quo approaches to drag-and-drop, we would need to manually construct drop targets for each of these target states. But given a render function, this work becomes unnecessary – we can instead run each state we wish to make accessible through the render function, generating three rendered racks, and extract the location of B in each rack (the *target positions*). A simple drag-and-drop interaction can then be generated from these positions: on drop, find the closest target position and transition into its corresponding state. This approach, which we call *model-driven dragging*, proves to be quite general. Figure 1 shows it in schematic form in a similar situation with multiple rows.

3.2 Composable Dragging

Through model-driven dragging, we can compute the most important information that drives a drag-and-drop interaction: target positions. However, this leaves unspecified a number of spatial and visual aspects of a drag interaction. Over what radius is a drop target active, and what happens if we drop further than this? Will the dragged element stick to the pointer during the drag, or will it sometimes spring into target positions? How will changes be smoothed over with animated transitions? Interface authors will be frustrated if they cannot control these aspects when desired.

We approach the challenge of declaratively and expressively specifying high-quality drag behaviors with the aid of *composability*. Our first step is identifying the abstraction of a *drag behavior*, which describes the behavior of a drag, frame by frame. It is a function that takes a pointer position and produces a *preview*, which specifies the rendered interface at this moment of the drag, and a *drop state*, which specifies what model state the interface should transition into if the drag were ended at this moment.

Developing declarative dragging, we discovered that a large variety of complex drag-and-drop behaviors found in the wild could be created from simpler behaviors. For instance, block-based programming environments often allow blocks to either be snapped into special ports or freely positioned on a canvas. This can be constructed from two sub-behaviors, a *snapping* behavior and a *free* behavior, using a Dragology combinator called ***whenFar***: the final behavior is denoted *snapping*`.`***whenFar***`(`*free*`)`. The *snapping* behavior is itself expressed using a combination of individual snap-points: `d.`***closest***`(`*snap1*`, `*snap2*`, ...)`. Together, these operators form a domain-specific language for describing drag-and-drop interactions. This language-oriented approach opens a large space of combinatorial possibilities and offers a new way to think and communicate about these interactions (as study participants note in §8.7).

4 Dragology: Motivating Examples

We now introduce Dragology, our library for declarative dragging, through two scenarios of use.

Scenario 1: Ring of Beads. Alex, a mathematics professor, is preparing a lecture on "permutations of cyclic structures" (colored beads on a ring). Alex wants to use interactive diagrams to make these abstract mathematical structures tangible, and decides to make some gadgets to support class discussion. Alex has some coding experience, but is not an expert at implementing interactions. First, they code up a non-interactive diagram which takes a model state like `{beads: ["A", "B", "C", "D", "E"]}` and draws beads in that order around a ring, as seen in Figure 2 Ⓐ. Alex wants to be able to drag the beads on the ring into different orders. They decide to use Dragology, and begin by attaching `dragologyOnDrag` attributes to the bead elements. When dragging a given bead, the *reachable model states* should be those where the bead is removed from `state.beads` and reinserted at each possible location. So in the code for bead `idx`, Alex writes:

```
dragologyOnDrag={() => {
  const newStates = state.beads.map((_, targetIdx) => {
    const newState = structuredClone(state);
    newState.beads.splice(idx, 1);
    newState.beads.splice(targetIdx, 0, bead);
    return newState; });
  return d.closest(newStates); }}
```

The first part uses JavaScript to generate new "reinserted" states. The last line combines these states into a drag spec using the ***closest*** operator, accessed through a `d` object Dragology provides.

This is the only code Alex needs to add to create the interaction shown in Figure 2 Ⓐ - Version 1. By itself, ***closest*** creates an aggressively "snappy" interaction: the dragged **A** always jumps

[3] Drags that are not (straightforwardly) repositional include dragging with a brush to make paint strokes, or dragging from a color swatch onto a shape to apply the color.

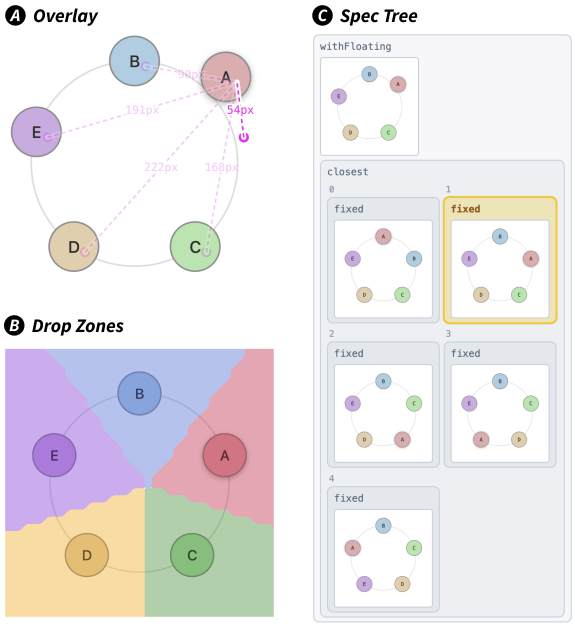

Figure 3: Three visualizers available to Dragology developers, shown in the context of Figure 2 Ⓐ - Version 2: Ⓐ Overlay, which in this case shows the pointer-target distances *closest* is consulting to pick the active branch. Ⓑ Drop Zones, which shows which sub-behavior is triggered by different pointer positions (sampled in a grid). Ⓒ Spec Tree, which shows how each operator in a drag spec contributes to the observed behavior. Here, it shows how *closest* picks the closest state and *withFloating* detaches the dragged element.

to the closest valid position, remaining still for the first part of the drag (1-2) and then swapping suddenly with **B** (3-4). Playing with this interaction, it feels strange to Alex that dragging a bead initially offers no visual feedback. So Alex changes the last line to `d.closest(newStates).withFloating()`. The *withFloating* modifier detaches the dragged element, making it follow the pointer directly. The resulting behavior is shown in Figure 2 Ⓐ - Version 2. Now **A** always follows the pointer (1-2) while the other beads move out of its way to preview where **A** will drop on pointer release (3-4). Like every other part of these interactions, **A**'s drop from floating into its final location is animated.

Scenario 2: Scheduler. The following week, Alex plans to lecture on a combinatorial optimization problem – assigning pre-scheduled events to as few rooms as possible without conflicts. Alex wants an interactive playground where students can try out this challenge. Alex represents the model state with a data structure:

```
{ meetings: [
  { start: 20, end: 100, room: 0 },
  { start: 60, end: 160, room: 1 }, ... ] }
```

As before, Alex's first step is to write a function that renders this data non-interactively. They represent meetings as bars along tracks representing rooms, shown in Figure 2 Ⓑ. To help students explore dimensions of scheduling, Alex wants to surface two interactions: dragging meetings between rooms to change assignments, and dragging meeting endpoints to change start and end times.

Like dragging beads, the first of these interactions is *discrete*: the dragged meeting should "snap" into one of a list of states Alex can enumerate. Alex implements this by attaching a `dragology` `OnDrag` attribute to each meeting bar, which enumerates states where the meeting's room number takes on different values. This determines drop targets. Next, Alex picks the Dragology operators that will shape the drag preview. *closest* is too constrained — it locks the meeting to its track until the pointer reaches a halfway point. *closest* + *withFloating* isn't constrained enough – the floating element can move left and right, even though this drag doesn't actually change the meeting's times. Alex settles on *between*, which lets the meeting slide in the convex hull of drop targets – in this case, a vertical track. This track is made visible by the Overlay visualizer (seen in the magenta box in Figure 2 Ⓑ). Dragology provides several visualizers, shown in Figure 3. If something isn't behaving the way Alex expects, they can switch these on and inspect the process that led to their observed behavior.

The second interaction, changing times, is different because time is a continuous numeric parameter. Alex can't make a state for every possible time. Instead, they use a drag operator called *vary*. On the "start" endpoint, they add:

```
dragologyOnDrag={() => {
  return d.vary(state, param("meetings", i, "start"), {
    constraint: (s) => [
      lessThan(0, s.meetings[i].start),
      lessThan(s.meetings[i].start, s.meetings[i].end - MIN_LENGTH),
    ]});}}
```

This reads: "Starting with the current `state`, vary the number at `state.meetings[i].start`, subject to the given constraint". As the endpoint is dragged, Dragology continuously updates this parameter to minimize the distance between the endpoint and the pointer position. Alex adds a similar attribute to the "end" endpoint. After testing this interface, Alex decides to inset the endpoints into the meeting bars rather than centering them, since this shows ranges more accurately. This requires a change to their render function, but the drag logic stays the same. It is expressed semantically, in terms of control over parameters, not fragile coordinate logic.

Mathematically, this scheduling problem is actually "coloring an interval graph". Alex adds a network representation with draggable nodes on the right, making mathematics tangible through two tightly synchronized representations.

5 Dragology: Implementation

Dragology is a library for the Web platform written in TypeScript. Dragology uses React [20] as a platform for functional views (§2.2). A developer uses Dragology by writing one or more Draggable[4] components: pure functions that turn a model state into a React node representing an SVG view.[5] Our decision to work directly on SVG, rather than higher-level representations, follows the principle of representational transparency established by Bostock et al.'s D3 [3].

[4]See Appendix A for a list of Dragology's core types, like Draggable.
[5]A React node is a lightweight data structure representing a tree of DOM nodes. A Draggable is not a React component – React features like "hooks" are not supported, and React components cannot be embedded inside of it.

To actually run a Draggable component as part of a larger interface, a developer hands it to the `<DraggableRenderer>` React component provided by Dragology. This component implements the "engine" that manages drag-and-drop interactions. When rendering, `<DraggableRenderer>` calls its Draggable, passing the current model state and receiving an SVG view tree in return. This tree is scanned for `dragologyOnDrag` attributes, which are replaced with DOM `onPointerDown` attributes which tell the engine to start a drag using the drag-spec callback provided. The SVG is then handed to React for final rendering.

Drag specs & behaviors. When a drag begins, Dragology calls the dragged element's `dragologyOnDrag` callback to obtain a DragSpec. This is an abstract syntax tree (AST) which is immediately compiled into a DragBehavior by an internal function ***dragSpecToBehavior***. A DragBehavior typically lasts for the duration of the drag (excepting chaining, see §6.3). On each frame, Dragology runs the DragBehavior on the pointer position, producing a *view* and a *dropState*. It displays the *view* and, if the pointer is released in this frame, transitions into the *dropState*.

A DragBehavior is generally a pure function. However, since they are implemented as JavaScript closures, it is possible for ***dragSpecToBehavior*** to initialize state that is shared between invocations. This enables path-dependent behavior, like ***closest***'s `stickiness` parameter that causes it to preferentially stay on the current branch.

Interpolation and animation. Dragology produces smooth interactions in large part by *interpolating views*. Dragology implements a ***lerpViews*** routine that interpolates between two SVG React nodes. When the two views are structurally identical (same tree of element types), ***lerpViews*** interpolates attribute-by-attribute, handling types like numbers, colors, transforms, and paths. When the views differ in their structure, interpolation becomes a challenge, as ***lerpViews*** must determine correspondences between nodes in the source and target. To resolve this, we follow React's use of keys.[6] While React's keys are local, identifying children of a single node, Draggable components use global IDs provided via the standard DOM `id` attribute. This extends interpolation to support structural changes where elements are re-parented in the view tree.

Dragology uses ***lerpViews*** in two situations: (1) running animated transitions between previews, and (2) ***between***, which interpolates between previews based on the pointer's position. If there are more than two target states in a ***between***, we construct a Delaunay triangulation of target positions and use the pointer's barycentric coordinates in a triangle to interpolate up to three views. We also support natural-neighbor interpolation [33], which produces smoother motion at the cost of interpolating more than three views at a time.

Numerical optimization. As introduced in §4, Dragology's ***vary*** primitive lets a continuous drag control developer-specified numeric parameters in the model state. It does this using numerical optimization. Every setting of specified numeric parameters determines a drop state. By rendering this drop state and extracting the dragged element's position, we can determine a target position and measure the distance between this position and the pointer. This determines an objective function, which we can feed to a nonlinear minimization algorithm. Dragology uses an implementation of COBYLA extracted from the JSXGraph library [13, 26]. While numerical optimization can present challenges in general, Dragology's use of it is fairly tame. Typical drags only control one or two parameters at a time, avoiding the pitfalls of high-dimensional spaces, and the logic of render functions frequently results in landscapes amenable to optimization. While our use of numerical optimization is limited, it plays an important role, allowing drags to target continuous manifolds of states just as easily and declaratively as discrete sets.

[6] https://react.dev/learn/rendering-lists#keeping-list-items-in-order-with-key

Primitives - produce base behaviors	
`state`	*all*
Use a state as a static drag-spec.	
`d.`***vary***`(state, paths, opts?)`	*4, 6.1-6.4, B.1-B.2*
Pull dragged element to pointer by varying numbers at `paths` in `state`.	
`d.`***dropTarget***`(targetId, state)`	*6.2, B.1*
Transition to a new state when dropped over a specified element.	
Combinators - compose behaviors	
`d.`***closest***`([spec1, spec2, ...], opts?)`	*4, 6.1-6.3, B.1-B.2*
Pick the behavior resulting in the smallest pointer-target distance.	
`d.`***between***`([spec1, spec2, ...], opts?)`	*4, 6.3-6.4, B.1*
Interpolate smoothly between behaviors inside their convex hull.	
`spec.`***whenFar***`(specFar, opts?)`	*6.1, 6.2, B.1-B.2*
Switch from `spec` to `specFar` when pointer is far from `spec`'s targets.	
`d.`***switchToStateAndFollow***`(state, id, spec?)`	*6.2, B.1-B.2*
Immediately switch to a new state and follow a new element.	
Modifiers - layer on additional features	
`spec.`***withFloating***`(opts?)`	*4, 6.1-6.2, B.1*
Detach dragged element and make it follow pointer.	
`spec.`***withSnapRadius***`(radius, opts?)`	*6.3-6.4*
Pull the preview towards the drop state when close enough.	
`spec.`***onDrop***`(func)`	*B.2*
Transform the drop state using a provided function.	

Figure 4: Dragology drag-spec operators discussed in this paper. Optional arguments are marked with '?'.

Operators. The DragSpec in a `dragologyOnDrag` callback is constructed by an interface developer using a domain-specific language of operators. §4 introduced a few of these: ***closest***, ***withFloating***, ***between***, and ***vary***. Figure 4 summarizes more, many of which will be described in §6. We have worked to make this set of operators generic and compositional. For instance, in an earlier version we had a `d.floating(`*states*`)` primitive that operated directly on states. We later realized that this behavior could be rebuilt as a modifier that could apply to any behavior – not just the original `d.`***closest***`(`*states*`).`***withFloating***`()`, but new useful combinations like `d.`***vary***`(...).`***withFloating***`()`.

Because all an operator does is construct a DragBehavior function, users of Dragology can implement their own operators. All of Dragology's operators could be implemented in this way, outside of the library.

6 Dragology: Demonstrations

This section presents four more interfaces built with Dragology, shown in Figure 2 Ⓒ-Ⓕ, highlighting further features and comparisons to status-quo approaches.

6.1 Lists in Lists

Figure 2 Ⓒ shows an interface where white tiles are sorted into lists, which can themselves contain nested lists, and where top-level lists are freely positioned on a canvas. Panels 1-3 shows a user dragging a tile out of a list to create a new top-level list. Panels 4-5 shows the user dragging this tile back into an existing list.

When an item (tile or list) is dragged, it will end up either freely positioned on the canvas or snapped in as a child of an existing list. This combination of free-form and structured dragging is found in applications like Scratch [29] and similar block-based programming environments. With Dragology, the free-positioning part can be defined as a continuous drag with ***vary*** and the structured snapping can be defined with ***closest***. We combine these with the combinator ***whenFar***: *snapping*.***whenFar***(*freePositioning*) means "use *snapping*, unless that results in a pointer-target distance beyond a certain threshold, in which case use *freePositioning*". The effect of these operators working together is shown in the Drop Zones visualizer (magenta box).

Although this example may seem simple, implementing it without Dragology is difficult and error-prone. We ran a preliminary experiment to see how existing frameworks fared. We found that asking an LLM coding agent to implement this functionality with status-quo libraries resulted in success rates of 0-40%, compared to 80% for Dragology. Code written for these other frameworks also took 87% to 103% more lines of code than Dragology. See Appendix C for details on our methodology and results.

6.2 Tactile Tessellations

Figure 2 Ⓓ shows an interface where a user can snap together tilings from triangles, squares, hexagons, and octagons. These shapes can be freely positioned on the canvas. When a shape comes close to an unused edge of an existing shape, it "snaps into" that edge, rotating and translating as needed. This is "free positioning with special snapping points" as seen in Lists in Lists (§6.1, ***whenFar***).

Shapes can be removed by dragging them into a "trash bin". Dragology cannot use its typical model-driven approach to infer this behavior, since the dragged element does not exist in the post-removal model state. This is a situation better handled by the status-quo concept of an explicit "drop zone", so Dragology provides ***drop Target*** – a behavior where dropping a dragged element onto another element specified by an id (like `"trash-bin"`) will transition to a specified drop state. This escape-hatch behavior can be combined with Dragology's "model-driven" operators – Tactile Tessellations simply includes it in a ***closest*** together with edge-to-edge snapping states.

New shapes can be created by dragging them from a toolbar. This presents similar problems for model-driven dragging: the shape we want to drag doesn't exist yet at drag start. We solve this with an operator called ***switchToStateAndFollow***. This behavior immediately transitions into a specified model state when the element is dragged – in this case, a model state where a new shape has been created. After this transition, the drag continues, but dragging a new element specified by a provided ID – in this case, the new shape's element. Dragology will consult the `dragologyOnDrag` attribute of this new element, which in this case will continue the drag using the canvas-dragging behavior described above.

Drags that create or destroy objects, such as those from the toolbar or into the trashcan, are not *a priori* "repositional" (§3.1). But Dragology operators like ***switchToStateAndFollow*** and ***drop Target*** allow us to handle these cases compositionally.

6.3 Animate Algebra

Figure 2 Ⓔ shows an algebraic expression in tree form. On the left, we start with $(((-(🍄)) + (⛅)) + (🍄))$, where nested parentheses represent nodes. Each of these nodes can be dragged to rewrite the expression according to certain laws. Panels 1-3 show $(⛅)$ being dragged upwards, commuting with $(-(🍄))$ according to the commutative law $A + B = B + A$. After this drag is completed, drags in multiple directions become available – not just dragging back the way we came, but also dragging $(⛅)$ *leftwards* to effect the associative law $(A + B) + C = A + (B + C)$ (shown in panels 3-5).

The interaction in this diagram is centered around *tracks*. From a source position, each possible destination determines a linear track. As the user drags, the closest track is identified and "locked onto". This design helps emphasize how dragging in different directions performs different rewrites. In this diagram's Dragology implementation, each track is some $trackSpec_i$ `= d.`***between***`([`*state*`,` $newState_i$`])`. These tracks are composed into a single spec with `d.`***closest***`([`$trackSpec_1$`,` $trackSpec_2$`, ...])`. The magenta box shows how the Overlay visualizer depicts the two tracks available to the dragged element from panel 5.

The rewrite in panels 1-3 and the rewrite in panels 3-5 drag the same node. This interaction would be most fluid if we could drag the node through both of these moves without putting it down. But by default, a drag in Dragology is determined by the spec provided at drag start with `dragologyOnDrag`, so performing a series of moves with a node would require dropping it and picking it up again at each position. Animate Algebra avoids this awkwardness using *chaining*, our term for transitioning model states *during* a drag. The ***withSnapRadius*** modifier, which is mainly used to visually snap onto a drop state, takes an optional argument enabling chaining on snap.

6.4 Twisted Trees

Figure 2 Ⓕ shows an order-preserving function f from a tree to itself: if A is a descendant of B, then $f(A)$ must be a descendant of $f(B)$. Nodes in the domain are represented with black circles and nodes in the codomain with grey ones. Each black domain node sits inside of the grey codomain node it is mapped to by f. The grey nodes expand and contract to fit the black nodes inside them.

Any black node can be dragged to any grey node; other black nodes may be pulled along if the order-preserving constraint demands it. Like Animate Algebra, this interface uses ***between***, but here every target state is placed in a single ***between***, allowing free-form motion in the convex hull of target positions (shown in the Overlay visualizer in the magenta box). As the pointer moves in two dimensions, the preview fluidly interpolates between rendered

states. Grey circles inch larger and smaller, anticipating the nodes that will move in or out of them if the user keeps moving in their direction. In this interface, users can "feel out" the consequences of different directions before they arrive at their destination, a form of "feedforward" [9, 35].

7 Study Methods

We ran a hands-on interview study with 13 participants to probe the following questions:

- RQ1. How approachable and easy-to-use is declarative dragging? How do practitioners react to this new paradigm?
- RQ2. What role if any do practitioners see for Dragology in their own projects and domains?
- RQ3. What does it feel like to explore the design space of interactions with declarative dragging?
- RQ4. How might declarative dragging affect what drag-and-drop interfaces are built?

Recruitment & participation. For this study, we sought participants with a background and ongoing interest in developing drag-and-drop interfaces. We recruited participants through individual outreach (7 participants) and broader outreach via social media and relevant online communities (6 participants). While participants were all experienced programmers, their backgrounds and interests varied. They included game developers, researchers in HCI, PL, and data visualization, educators, authors of explorable explanations [7, 36], and a museum exhibit developer. Reported ages ranged from 21-39 and three of our participants were women.

Structure. Study sessions were 90-120 minutes, conducted over video conferencing. They began with a semi-structured 20-30 minute interview, where we asked participants about their prior experiences with drag-and-drop.

The next 60-90 minutes consisted of a pair-coded tutorial, with participants controlling the mouse and keyboard. We guided participants through six example interfaces designed to introduce Dragology in small steps, detailed in Appendix D. Two of these examples were complete. The other four had complete render functions, but with drag-and-drop interactions missing. For these, we asked participants to try implementing the interaction themselves, with us providing assistance on request or as necessitated by time constraints. Participants worked on their own computers, in their own preferred development environments, using whatever AI assistance they desired. We encouraged participants to experiment as they went through the examples.

After working through these six examples, we directed participants to the Dragology demos page for further exploration and discussion. After the study, we sent participants an exit survey.

Limitations. Our study has standard limitations for qualitative interview research. The recruitment process biased our participant pool towards people with prior interest in HCI and PL, compared to a broader programmer population. Sessions were single-exposure in guided tutorial format, offering limited evidence of independent learning processes or adoption. The study authors were present during all sessions, introducing potential social desirability bias.

8 Study Results

We annotated interview recordings and exit surveys using reflexive thematic analysis [4, 5], producing over 500 codes. Our approach was primarily inductive, coding from participants' experiences, with our research questions providing a scaffold. Codes were clustered into candidate themes and refined through discussion, with the most salient themes presented below.

8.1 Drag-and-drop is hard and often avoided

Each participant characterized implementing drag-and-drop as difficult, tedious, or both. Many reported that this directly shapes what gets built, causing them to avoid entire interaction categories. P03 said Dragology "solves a problem that I have struggled with so much that usually when I have to do a drag and drop interaction, I just don't bother".

Drag-and-drop is cross-cutting. P09 reports that "the main issue with the code is that it's not very local. You're coding a button, but it has to know all about the drag-and-drop system." P03 notes "multiple continuous processes, some of them user driven and some of them animation driven, overlap [in ways] that you just don't run into with other kinds of interfaces." P09 described the core problem: "drawing needs to know how to go from the state to the drawing, and computing has to know how to go from the drawing to the state, and usually those are in separate parts of the code".

Drag-and-drop complicates state. Some participants manage complexity via view/state separation, which drag-and-drop complicates: "the separation between updating your state and rendering [...] prevents bugs. But that means that your state needs to be rich enough to capture things like intermediate states where I am holding a [draggable element]" (P06).

Another cluster of pain points relate to coordinate system math, including animation, grab anchors, and drop target geometry. P10: "I just don't want to have to deal with the physics of it".

8.2 Ease of use: Concise code, less math (RQ1)

Concise code. Participants expressed surprise at how little code was needed using Dragology. P06 said "what's jumping out to me is like: where is the code?" and later compared their own reorderable list side-by-side, finding the Dragology version "nicer and much less code". P04 found that with "one to five lines, you can transform something from being static to something that's very interactive and has very reasonable default behaviors", with P03 similarly noting that "the default behaviour of the library requires very little code to deliver a great-feeling result". P04 summarized in the exit survey: "the code I was writing was almost entirely about specifying the specific interaction I wanted. So much boilerplate and incidental complexity was removed".

No math. Dragology eliminates much of the manual geometric computation traditionally required for drag-and-drop. P04 remarked of the rotary dial: "I think oftentimes when you get into these SVGs and polar coordinates and all this stuff, it can be much more annoying to implement. But this felt as easy as implementing your bog standard horizontal slider".

Composition reduces friction. With a compositional API, adding behaviors (floating, snapping, transitions) often does not require restructuring existing code. P06 found "that's very cool that I could just tack on this call and get what I wanted instead of changing the state". P01 contrasted this with existing approaches: "all the libraries I see are 'here's the behavior and we're going to give you a bunch of flags'. But they don't give me control over the behavior".

8.3 Learnability: State space stresses (RQ1)

Model-driven dragging requires a shift in perspective, described as "brain-twisting" (P12) and "brain-breaking" (P7). Study trajectories were generally smooth through the switch tasks, with a sharp cliff at the reorderable list. P03 remarked that "this feels complicated and mechanical in a way that nothing else has yet ... the illusion of simplicity has been shattered for me."

Declarative dragging requires developers to reason about sets of reachable states. This demand has no direct analogue in traditional user interface development. P09 described the contrast: "I am used to thinking only about [single] state transformations 'if I drop this here, then the new state will be...' instead of states 'by dragging this around, I can reach any of these N states'. Thinking about 'all reachable states' felt a bit unnatural at first". For the switch tasks, this unfamiliarity was masked by simplicity; listing a handful of concrete states does not feel like "state space reasoning".

The issue became acute with the reorderable list: participants must now abstractly compute sets of states, reasoning from the perspective of a single element while producing whole-list state objects. P13 described the difficulty as moving "from the frame of 'which DOM guy am I sticking this dragology on' to 'how am I assembling the state tree'". This caused some participants to misstep, trying to enumerate all permutations rather than the "local" successor states reachable by dragging a given element. As P12 put it, it was "hard to think about the localized transition rules from the current state to the possible next states".

Participants with functional programming backgrounds (P02, P06, P10, P11) seemed more inclined to accept the complexity as worthwhile, with P02 saying "as someone who likes functional programming [...] I think it's good that it forces me to think about the space of possible states". Several participants described the model as as a natural extension of React's functional paradigm.

Enumerating states isn't very declarative. State construction – the act of programmatically generating the set of reachable states – was identified by most participants as the sharpest point of friction. For simple discrete cases it is trivially concise; P01 said of the switch "I can't imagine a simpler [API] interface here, because I have to list the states I can get to". But for collections, it becomes a combinatorial problem, aggravated by JavaScript's poor ergonomics around immutable data structures. P04 felt "it sort of exploded the complexity all of a sudden, whereas everything else up to this point had been so elegant". Some felt state construction was inconsistent with the rest of the library's declarative spirit, with P05 saying that "instead of specifying [...] 'go generate all of the concrete states' I want to do something more declarative, [...] higher level."

The details of state construction are arguably orthogonal to Dragology: the library takes states as input and is agnostic about how they are produced. But since the model-driven approach requires enumeration, the friction is experienced as a Dragology problem, and perhaps the biggest conceptual hurdle to adoption. Immutable-state libraries like Immer [38] may partially address the mechanical burden, and participant P10 proposed Prolog-style relational specification of reachable states, suggesting an avenue for future work.

Magic considered harmful/helpful. The term "magic" appeared often, both positively and pejoratively. P01 said "I have no idea what kind of horrors you have underneath to actually make this work, but it feels very nice as a user", while P11 found the compositional style "very cool at a high level but maybe a bit mystical at times". These concerns were partially allayed by the visualizers (Figure 3), which make speculative rendering visible.

8.4 Quality: Polished out of the box (RQ1, RQ4)

Automatic interpolation of non-dragged visual properties (colors, positions, sizes) consistently surprised participants. P01 noted "I did not have to write color interpolation, I did not have to write snap, I did not have to write the interpolation between these points". P06 remarked that the result "feels very fancy. It feels like the switch that someone paid money to sit down and carefully hand-roll", while P13 enjoyed that "the satisfying interaction drops out for free from this state tree". P06 compared Dragology's reorderable list directly against their own hand-rolled version and found the latter "really very jumpy", whereas "this one is like smooth". P01 acknowledged features they wished they could provide: "your transitions and your ghosting are impressive. I just don't do them. They're just too much work". Participants argued that these dimensions of interaction are not simply cosmetic, with P02 saying "I don't think it's a shiny decoration. I'm of the opinion that it's very important to give intuition through visual feedback".

8.5 Exploration: Play and discovery (RQ3, RQ2)

After grasping the core concepts, Dragology makes exploration cheap, with small code changes producing meaningful behavioral changes. Participants described the library as "promoting play" (P03) and forming a "playground to try out different combinations" (P05). P06 found it "much easier to add interesting interactions, almost as drastic as the difference between high-level and assembly-level programming".

Most participants experimented spontaneously. P09 swapped `translate` for `scale` on the switch, producing a switch that got bigger and smaller, and created parabolic slider paths and cycloid curves on the rotary dial. P12 added `Math.sin` to a slider's Y value, producing a wavy path, reporting "surprise, joy, and elation".

Some participants discovered designs through exploration rather than specification. P06 iterated through four combinator configurations on the reorderable list, arriving at a preferred design not previously envisaged. They contrasted this with prior experience, where "every trick I add makes my code far more complicated, so I'm reluctant to add them", whereas in Dragology, tricks compose cheaply. P10 put it this way: "Once the states (the what) is defined, playing around with how the drag interaction could be in between was much less of a stressful concern since I didn't worry about

screwing the end result". Separating 'what' from 'how' de-risks experimentation.

8.6 A niche: Animating abstractions (RQ4, RQ2)

Dragology seems to appeal to an underserved intersection: developers who want to add tangible interactivity to complex, idiosyncratic structures, but lack either the specialized knowledge or engineering resources for hand-rolled drag-and-drop. P02, a theorist whose main activity is logic rather than interface engineering, remarked "it's very useful in my specific position, because I don't have a lot of time to do some engineering to have very nice usable interfaces. And I think this library gives me a very declarative, very quick way to have good defaults with nice animations". P10, who is building mathematical explorables, expressed that "this is reinvigorating! This makes me so excited to continue work on [my own project], that I don't have to recreate new primitives [...] for each visual dialect out of things like `onDragStart`".

Enthusiastic responses spanned domains — programming tools, mathematical explorables, games, data visualization — suggesting what P13 called a "weirdly generalizable niche" not well catered to by existing approaches. Interfaces that turn abstract mathematical structures into fluidly interactive materials were a core motivation for developing Dragology. P13 said of Twisted Trees (§6.4): "This is a widget that I've never seen before, so I'm super excited that it is expressible through this library... This is what I would like to be able to do with Legos or K'nex or whatever, except the physical structures keep bumping into each other and breaking and these don't".

8.7 Dragology as language or paradigm (RQ4)

While Dragology's spec language was developed as a tool for constructing interfaces, formal languages can play a dual role as a "tool of thought" [18] and communication. P13 remarked that the spec language is "itself is a very cool contribution... having the names for 'here are ways of interpreting a user in the middle of a gesture'." P12 agreed, describing Dragology as "giving you a way to concretely talk about user interfaces".

Beyond the library as artifact, other participants said they expected D2 to influence how they build drag-and-drop interactions. P01 said that "it's definitely making me think about how I might implement my diagrams using some of these ideas", with P09 expecting that they "will be extracting some of this into my own code". In their exit survey, P05 wrote that "the idea of getting all possible end states as the guidance for [drag-and-drop] behaviors is a wow factor for me. I kept thinking about it when I coded my interface with [drag-and-drop] features later".

9 Limitations & Future Work

Limits on drag interactions. Dragology is centered on repositional drag-and-drop (defined in §3.1). As examples like Tactile Tessellations (§6.2) show, our composable behaviors can reach further than this to support patterns for object creation and destruction. But interactions not involving repositioning a persistent element, such as drawing gestures, scrubbing, or other pointer-path-based interactions, can be awkward or impossible to express.

Dragology has other limitations. It offers no principled model for velocity- or path-dependent drag effects, though drag behaviors have simplistic technical means to implement these. Drop states in Dragology must be spatially unambiguous; when two drop targets coincide, the system cannot disambiguate intent and throws an error. Although Dragology supports touch devices, it can only handle one touch at a time; multi-touch is a natural avenue for future work, especially as a way to resolve some cases of spatial ambiguity.

Beyond SVG. Declarative dragging requires (1) speculatively rendering candidate states, (2) extracting visual attributes from those renders, and (3) interpolating between those attributes. Dragology takes advantage of SVG's explicit positioning model, encoding positions via transform attributes, to simply and cheaply carry out all three. This is particularly important for the ***vary*** operator, where the renderer is invoked many times per frame. Extending this to generic HTML requires DOM layout to run prior to position extraction. This is feasible for discrete behaviors (akin to the FLIP animation technique [19]) but may be prohibitively costly for continuous behaviors. Platforms without element trees, like WebGL and Canvas, are not by themselves natural settings for our techniques. Study participants mentioned other platform constraints as adoption barriers, including React lock-in.

Performance. The cost of speculative rendering scales with the number of reachable states, incurring quadratic cost in the reorderable-list case where an ad-hoc imperative approach would be linear. Avoiding this cost in the most general case is difficult, though specific cases may permit speed-ups, such as those where the dragged element can be speculatively rendered in isolation, or those where layout can be incrementalized.

10 Conclusion

We present *declarative dragging* (D2), a new way to build drag-and-drop interfaces, combining *model-driven* and *composable* techniques to attain expressive breadth and declarative simplicity. Study participants found that our prototype library Dragology promoted exploration and produced polished results with surprisingly little code. Our demos and study suggest D2 can animate inert abstract structures into tangible, interactive material – enabling domain experts across eclectic fields to bring strange new interactions into being.

References

[1] Sean Barrett. 2005. Immediate Mode GUIs. *Game Developer* 12 (Sept. 2005), 34–36. https://web.archive.org/web/20210825/https://www.gdmag.com/

[2] Matthew T Beaudouin-Lafon, Devamardeep Hayatpur, Arvind Satyanarayan, and Haijun Xia. 2026. Belidor: A Specification Language for Operationalizing Structural Analogies Between User Interfaces. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems* (Barcelona, Spain) *(CHI '26)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3772318.3791613

[3] M. Bostock, V. Ogievetsky, and J. Heer. 2011. D³ Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (Dec. 2011), 2301–2309. doi:10.1109/TVCG.2011.185

[4] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3, 2 (Jan. 2006), 77–101. doi:10.1191/1478088706qp063oa

[5] Virginia Braun and Victoria Clarke. 2012. Thematic analysis. In *APA handbook of research methods in psychology, Vol 2: Research designs: Quantitative, qualitative, neuropsychological, and biological*, Harris Cooper, Paul M. Camic, Debra L. Long,

$$\begin{aligned}
\text{Draggable} &= \text{DraggableProps} \to \text{View} \\
\text{DraggableProps} &= \left\{ \begin{array}{r} \mathit{state} : \text{ModelState} \\ \mathit{d} : \text{DragSpecBuilder} \\ \mathit{draggedId} : \text{string} \mid \text{null} \\ \mathit{setState} : \text{ModelState} \to \text{void} \\ \cdots \end{array} \right\} \\
\text{DragBehavior} &= \text{DragFrame} \to \text{DragResult} \\
\text{DragFrame} &= \{\, \mathit{pointer} : \text{Vec2} \,\} \\
\text{DragResult} &= \left\{ \begin{array}{r} \mathit{preview} : \text{View} \\ \mathit{dropState} : \text{ModelState} \\ \mathit{gap} : \text{number} \\ \cdots \end{array} \right\} \\
\text{DragSpec} &= \{\, \mathit{type} : \text{"fixed"},\ \mathit{state} : \text{ModelState} \,\} \\
&\mid \{\, \mathit{type} : \text{"closest"},\ \mathit{specs} : \text{DragSpec}[\,] \,\} \\
&\mid \cdots \\
\mathit{dragSpecToBehavior} &: \text{DragSpec} \times \text{DragInitContext} \to \text{DragBehavior} \\
\text{DragInitContext} &= \left\{ \begin{array}{r} \mathit{draggable} : \text{Draggable} \\ \mathit{draggedPath} : \text{string} \\ \mathit{draggedId} : \text{string} \mid \text{null} \\ \mathit{anchorPos} : \text{Vec2} \\ \mathit{pointerStart} : \text{Vec2} \\ \mathit{startState} : \text{ModelState} \end{array} \right\}
\end{aligned}$$

Figure 5: Core type definitions for Dragology's component and drag behavior system.

A. T. Panter, David Rindskopf, and Kenneth J. Sher (Eds.). American Psychological Association, Washington, DC, 57–71. doi:10.1037/13620-004

[6] Magnus Carlsson and Thomas Hallgren. 1993. Fudgets. In *Proceedings of the conference on Functional programming languages and computer architecture*. 321–330. doi:10.1145/165180.165228

[7] Nicky Case. 2015. Explorable Explanations. https://explorabl.es/.

[8] Claudéric Demers. 2020. dnd kit. https://dndkit.com/.

[9] Tom Djajadiningrat, Kees Overbeeke, and Stephan Wensveen. 2002. But how, Donald, tell us how? on the creation of meaning in interaction design through feedforward and inherent feedback. In *Proceedings of the 4th conference on Designing interactive systems: processes, practices, methods, and techniques (DIS '02)*. Association for Computing Machinery, New York, NY, USA, 285–291. doi:10.1145/778712.778752

[10] Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*. ACM, Amsterdam The Netherlands, 263–273. doi:10.1145/258948.258973

[11] Epic Games. 2024. Blueprints Visual Scripting in Unreal Engine. https://dev.epicgames.com/documentation/en-us/unreal-engine/blueprints-visual-scripting-in-unreal-engine. Accessed: 2026-03-31.

[12] Evan Czaplicki. 2012-2020. Elm. (2012-2020). http://elm-lang.org.

[13] Michael Gerhauser, Bianca Valentin, and Alfred Wassermann. 2010. JSXGraph–Dynamic Mathematics with JavaScript. *International Journal for Technology in Mathematics Education* 17, 4 (2010), 211–215. https://www.learntechlib.org/p/109512/

[14] Thomas R. G. Green and Marian Petre. 1996. Usability Analysis of Visual Programming Environments: A Cognitive Dimensions Framework. *J. Vis. Lang. Comput.* 7, 2 (1996), 131–174. doi:10.1006/jvlc.1996.0009

[15] David Harel. 1987. Statecharts: a visual formalism for complex systems. *Science of Computer Programming* 8, 3 (June 1987), 231–274. doi:10.1016/0167-6423(87)90035-9

[16] Paul Henschel. 2018. react-spring. https://www.react-spring.dev/.

[17] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. 1985. Direct Manipulation Interfaces. *Hum. Comput. Interact.* 1, 4 (1985), 311–338. doi:10.1207/s15327051hci0104_2

[18] Kenneth E. Iverson. 1980. Notation as a tool of thought. *Commun. ACM* 23, 8 (Aug. 1980), 444–465. doi:10.1145/358896.358899

[19] Paul Lewis. 2015. *FLIP Your Animations.* https://aerotwist.com/blog/flip-your-animations/ Blog post.

[20] Meta. 2013. React. https://react.dev/.

[21] Mozilla Developer Network. 2024. HTML Drag and Drop API. https://developer.mozilla.org/en-US/docs/Web/API/HTML_Drag_and_Drop_API.

[22] B.A. Myers, D.A. Giuse, R.B. Dannenberg, B.V. Zanden, D.S. Kosbie, E. Pervin, A. Mickish, and P. Marchal. 1990. Garnet: comprehensive support for graphical, highly interactive user interfaces. *Computer* 23, 11 (Nov. 1990), 71–85. doi:10.1109/2.60882

[23] Brad A. Myers. 1990. A new model for handling input. *ACM Transactions on Information Systems* 8, 3 (July 1990), 289–320. doi:10.1145/98188.98204

[24] Stephen Oney, Brad Myers, and Joel Brandt. 2014. InterState: a language and environment for expressing interface behavior. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*. ACM, Honolulu Hawaii USA, 263–272. doi:10.1145/2642918.2647358

[25] Matt Perry. 2019. Motion. https://motion.dev/.

[26] M. J. D. Powell. 1994. A Direct Search Optimization Method That Models the Objective and Constraint Functions by Linear Interpolation. In *Advances in Optimization and Numerical Analysis*, Susana Gomez and Jean-Pierre Hennart (Eds.). Springer Netherlands, Dordrecht, 51–67. doi:10.1007/978-94-015-8330-5_4

[27] Miller Puckette. 1991. Combining Event and Signal Processing in the MAX Graphical Programming Environment. *Computer Music Journal* 15, 3 (1991), 68–77. doi:10.2307/3680767

[28] Alex Reardon. 2024. Pragmatic drag and drop. https://atlassian.design/components/pragmatic-drag-and-drop/.

[29] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Communications of the ACM (CACM)* (2009).

[30] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Trans. Vis. Comput. Graph.* 23, 1 (2017), 341–350. doi:10.1109/TVCG.2016.2599030

[31] Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. 2014. Declarative interaction design for data visualization. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*. ACM, Honolulu Hawaii USA, 669–678. doi:10.1145/2642918.2647360

[32] Schachman, Toby. 2015. Apparatus. http://aprt.us/.

[33] Robin Sibson. 1981. A Brief Description of Natural Neighbour Interpolation. In *Interpreting Multivariate Data*, Vic Barnett (Ed.). John Wiley & Sons, New York, 21–36.

[34] Noi Sukaviriya, Srdjan Kovacevic, James D. Foley, Brad A. Myers, Dan R. Olsen, and Matthias Schneider-Hufschmidt. 1994. Model-based user interfaces: what are they and why should we care?. In *Proceedings of the 7th annual ACM symposium on User interface software and technology - UIST '94*. ACM Press, Marina del Rey, California, United States, 133–135. doi:10.1145/192426.192479

[35] Jo Vermeulen, Kris Luyten, Elise van den Hoven, and Karin Coninx. 2013. Crossing the bridge over Norman's Gulf of Execution: revealing feedforward's true identity. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. Association for Computing Machinery, New York, NY, USA, 1931–1940. doi:10.1145/2470654.2466255

[36] Bret Victor. 2011. Explorable Explanations. http://worrydream.com/ExplorableExplanations/.

[37] Guillermo Webster. 2016. g9: Automatically Interactive Graphics. https://omrelli.ug/g9/.

[38] Michel Weststrate. 2017. Immer. https://github.com/immerjs/immer.

A Types

Figure 5 collects Dragology's core types. The main developer-facing entry point is the Draggable component type, a pure function from DraggableProps (which includes model state) to a view. ***dragSpecToBehavior*** compiles a DragSpec together with a DragInitContext to produce a DragBehavior. DragSpecs, whose constituent operators are collected in Figure 4, are the core forms of the Dragology DSL. See §5 for more details on the implementation.

B More Demos

B.1 Drag-Spec Designer

Figure 2 ❻ shows a programming environment for assembling Dragology drag-specs, made with Dragology. The testbed interface, shown in the "preview" at right, consists of a knob that can be dragged between three positions. It is running live, and can be tested

as the spec is composed. At first, the "active spec" port on the left is empty, and the knob cannot be dragged at all. As the user drags blocks representing operators and states into this port, and into nested ports below, they compose a drag spec. Panels 1-3 show a user dragging a ***between*** block with some state diamonds towards the next open port of a ***closest*** block. As the ***between*** block approaches, the port expands to indicate that it will accept the block on drop. This is implemented with a `d.closest(...).withFloating()` pattern, like Version 2 of Alex's beads (§4). These smooth animations use ***lerpViews***'s support for interpolating SVG paths.

After the user snaps in the ***between***, the active spec will read:

```
d.closest([d.between(A, B), d.between(A, C)])
```

This is essentially the "multiple-tracks" pattern used in Animate Algebra. The magenta box in the lower-right shows how the Overlay visualizer displays these tracks as the user tests their spec.

B.2 Nodes and Noodles

Figure 2 ⓗ shows a toy node-based programming interface, akin to Max/MSP [27] or Unreal Engine's Blueprints [11]. (Unlike DragSpec Designer, this is just an interface experiment – it doesn't run anything.) Nodes can be moved on a canvas. Wires connecting them can be created by dragging from ports on nodes, as seen in panels 1-3. Wires can have a single "loose end" on the canvas, as shown in panel 4. But we do not allow wires to have two loose ends: panels 4-6 show the attached end of this wire pulled out of the port and then released, resulting in the wire fading out of existence. This drag behavior uses the ***onDrop*** operator, which runs a function on drop to transform the drop state. Here, this function checks if the dragged wire has no attached ends, and if so, removes it from the state. Like ***switchToStateAndFollow*** and ***dropTarget*** (discussed in §6.2), ***onDrop*** opens up possibilities beyond positional drags.

C Framework Comparison Experiment

In §6.1, we summarized a preliminary experiment that compared a coding agent's performance using Dragology to reproduce the Lists in Lists example to its performance using prior frameworks. Here we describe our methodology and results in more detail.

Methodology. We tested seven libraries/approaches: Dragology, Motion [25], plain React, vanilla TypeScript without React, Pragmatic drag and drop [28], dnd kit [8], and react-spring [16]. In all conditions, the coding agent was given code for the Lists in Lists demo as a static React component without interaction, together with a detailed specification of a drag-and-drop interaction for it – complete with descriptions of what should be shown as a preview during drag and what transitions should be animated. The agent was further instructed to use the condition's specific library to implement the specification.

In non-Dragology conditions, the agent was allowed to access whatever resources it wished. In the Dragology condition, we instructed the agent not to access any resources outside of the Dragology library itself (including its types and documentation), and an "AGENTS.md" file we prepared, outlining how to work with Dragology and pointing out common "gotchas". This file contained short code snippets illustrating Dragology's mechanisms, but nothing approaching the complexity of the task. While there is a risk we may have still somehow "overfit" our instruction to the agent for this particular task, this possibility must be balanced against the fact that the agent has access to significant resources (likely including training data) guiding it in its use of the established libraries.

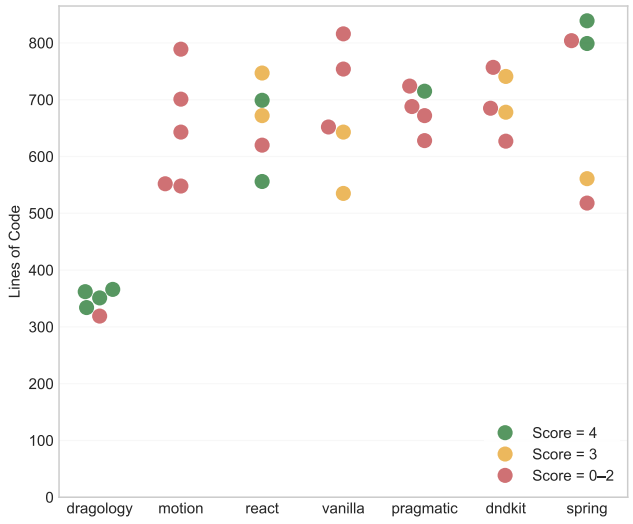

Figure 6: Results of a preliminary experiment comparing performance of an agent implementing a specification using different libraries/approaches. Trials are marked by dots, colored by the score we assign to the agent's implementation.

In each trial, we asked an LLM coding agent (Claude Code v2.1.85, running Opus 4.6 (1M context) with medium effort) to implement the specification. We then evaluated the implementation along two dimensions. First, we scored behavior on a scale from 0 to 4:

0. **Nothing is visible or nothing is interactive.**
1. **The interaction is not entirely functional.**
   e.g. sporadic crashes, positioning bugs
2. **The interaction is functional but has glaring artifacts.**
   e.g. distracting glitches on transitions, incorrectly laid-out previews
3. **The interaction is functional but has minor artifacts.**
   e.g. missing animations on transitions
4. **The interaction fully meets the specification.**

Second, we ran "prettier"[7] on the code to standardize its formatting and ran "cloc"[8] to count lines of code not including blanks and comments. To avoid the challenge of providing feedback to the agent in an unbiased and systematic way, we only judged the agent's first response – these results reflect "one-shot" performance. We ran five trials per library.

Results. See Figure 6. While Dragology attains the maximum score of 4 in 4 of 5 trials, other libraries attain this level in as few as 0 and as many as 2 trials out of 5. If we relax our threshold to a score of 3 ("minor artifacts"), the "plain React" condition also reaches the 4/5 level. The agent's Dragology implementations average 346 LOC,

[7] https://prettier.io/
[8] https://github.com/aldanial/cloc

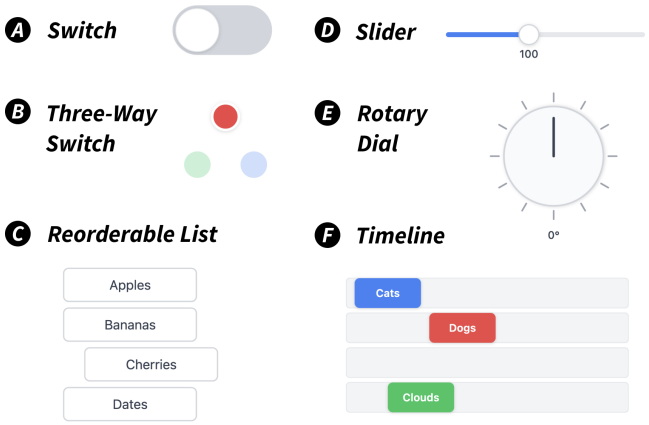

Figure 7: The six example interfaces used in our study.

while other libraries average 647 to 704 LOC, 87% to 103% more than Dragology.

Discussion. Dragology makes building interfaces like the Lists in Lists demo quite easy. Our goal with this study was to convey a rough sense of how much more difficult this task is using status-quo approaches, with a coding agent as a proxy for human developers. Naturally, this study is no substitute for a comprehensive evaluation with human participants.

D Study Examples

As discussed in §7, our study was centered around a tutorial which guided participants through six interfaces made (or waiting to be made) with Dragology. These examples are shown in Figure 7, and we provide short descriptions below.

A *Switch.* This was provided with a pre-built interaction, and served as a minimal introduction to Dragology. The switch has two states, and uses ***between*** to slide between them, with the background transitioning from grey to green.

B *Three-Way Switch.* This was provided without an interaction. It has three distinct states in a triangle. If the participant adds the three states to a ***between***, they will obtain a switch where the knob can go anywhere in the triangle between them. The knob itself will interpolate colors as it does so. Time permitting, we gave participants a further challenge: program the switch so that, starting at a given position, you can only move between that position and the next position clockwise, until you release the pointer.

C *Reorderable List.* This was provided without an interaction. We guided participants through (1) thinking about how to conceive of this interface in our "model-driven" framework, and (2) implementing it in Dragology, at first using the ***between*** construct they already know. After reflecting on this experience, we then introduced the participant to more combinators to create more behaviors, often prompted by feedback they give us about interaction styles they might prefer. (Most often, we introduced ***closest*** and ***withFloating***.)

D *Slider.* This was provided with a pre-built interaction, and served as a minimal example of continuous dragging with ***vary***. It often sparked experiments to probe the limits of ***vary***, such as making the slider knob move along more complex curves.

E *Rotary Dial.* This was provided without an interaction. It is a straightforward variation of Slider, but helps to convey the versatility of ***vary***.

F *Timeline.* This was provided without an interaction. It is a simple cartoon of a multi-track audio or video editor where clips can be moved along and between tracks. Movement along tracks is continuous, but movement between tracks is discrete. Achieving this combination requires composing operators more deeply than seen before, constructing a ***vary*** for each track and composing them using ***closest***.