Belidor: A Specification Language for Operationalizing Structural Analogies Between User Interfaces

Matthew T Beaudouin-Lafon
University of California San Diego
La Jolla, California, USA
mbeaudouinlafon@ucsd.edu

Devamardeep Hayatpur
University of California, San Diego
La Jolla, California, USA
dshayatpur@ucsd.edu

Arvind Satyanarayan
CSAIL
MIT
Cambridge, Massachusetts, USA
arvindsatya@mit.edu

Haijun Xia
University of California, San Diego
La Jolla, California, USA
haijunxia@ucsd.edu

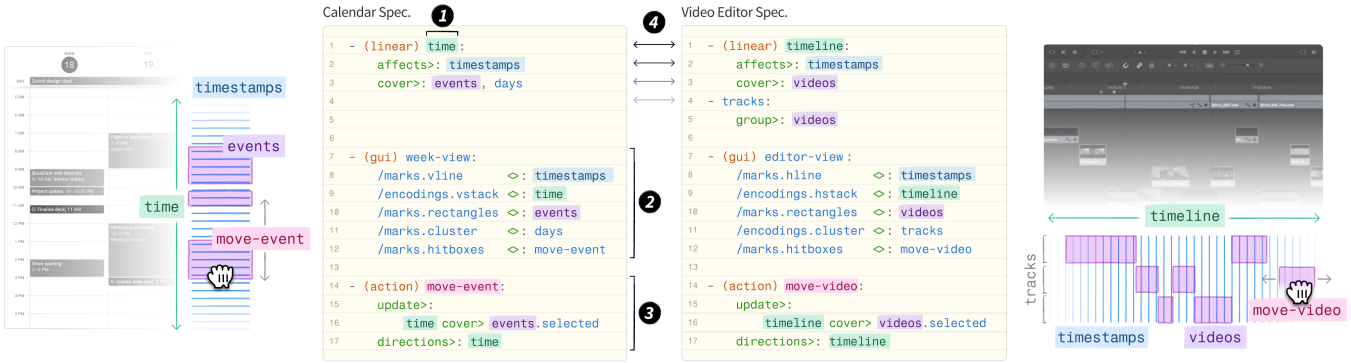


Figure 1: Belidor specifications describe the structure underlying user interfaces. Events in a calendar and videos in a video editor are both organized by their respective temporal structures (❶). Conceptual objects and structures can be presented by mapping them to the view's objects (marks) and structures (encodings) (❷). Structures also characterize interaction: for example, events/videos are directly manipulated along the time/timeline structure (❸). Belidor specifications help compare and contrast user interfaces, in this case by aligning the parts of the calendar/video editor specifications that overlap (❹), highlighting the analogy between the two user interfaces.

Abstract

We present Belidor, a text notation that describes the structure underlying user interfaces (UIs). Belidor's relational model emphasizes how structures, such as the temporal order of text messages, cut across an interactive system's conceptual model, user-facing presentation, and interactive behavior. We demonstrate Belidor's expressive power with a gallery of examples spanning GUIs (eg. messaging, video editors), screen readers, and hardware devices.

Belidor serves as an effective representation for structural analogies between user interfaces (eg. between calendars and video-editors). In contrast, prior work relied on visual UI representations and therefore prioritized visual style transfer. In three case studies, we show how Belidor can reveal analogies, help transfer ideas between user interfaces, and describe design patterns as analogies We discuss the implications of representing the structure of interactive systems for designers and developers, and envision how Belidor might support "structural design moves" for interface designers.

CCS Concepts

• **Human-centered computing** → **HCI theory, concepts and models**; **User interface design**.

Keywords

User Interface, Specification Language, Analogy, Structural Design

ACM Reference Format:
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 (CHI '26), April 13–17, 2026, Barcelona, Spain.* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/3772318.3791613

CHI '26, Barcelona, Spain

ACM ISBN 979-8-4007-2278-3/26/04
https://doi.org/10.1145/3772318.3791613

1 Introduction

Creative practitioners draw on existing work to inform their own. In interface design, borrowing patterns from existing user interfaces, like sidebars and carousels, prevents designers from inventing idiosyncratic solutions to solved problems. More interestingly, borrowing abstract aspects from a user interface can result in innovative designs. For example, Masson et al. designed Textoshop, a text editor that draws inspiration from an image editor with interaction techniques like "resizing" text selections, which rewrites the sentence in more or fewer words to fit the allocated space [48]. Their design is based on the analogy of "words as pixels, sentences as regions, and tones as colors." How can we operationalize analogies even between seemingly distant interfaces?

Prior work in HCI uses machine learning techniques on screenshots [11, 33, 38, 44, 66, 67] and website Document Object Models (DOMs) [5, 6, 41] to compute analogical mappings between user interfaces. Although the resulting analogical transfers are impressive, their scope is limited to surface-level analogies like visual transfer of style and layout (Section 2.2). This is because the visual representation of the application can only diffusely encode its underlying semantics. While analogies based on visuals can help designers improve the look of their user interface, they scarcely help iterate on its functionality—a far cry from the transfer of interaction techniques seen with Textoshop.

We contend that we need a better representation of interactive systems to support analogies between superficially different user interfaces. A similar story played out in the context of information visualization. With the Grammar of Graphics (GoG) Wilkinson sought to provide a language for charts that is both *expressive*, covering "every statistical graphic [he] had ever seen", and *parsimonious*, relying on composing only a few conceptual constructs[1] [65]. Defining deeper semantics helped compare charts: Wilkinson argues that the GoG explains *"why histograms are not bar charts and why many other graphics that look similar nevertheless have different grammars"*. Therefore, appropriate domain semantics can turn visually subtle differences into an obvious distinction, and seemingly different artifacts into surprising analogies.

To this end, we present Belidor,[2] a domain-specific language to represent interactive systems. Inspired by cognitive theories of analogy (Section 2.1), our key insight is to focus on how user interface objects relate to one another by surfacing their underlying *structure* as a first-class concept. For example, the linear structure of time affects text-messages, and a hierarchical structure organizes files and folders. By reifying these structures, Belidor affords direct comparison between abstract aspects of user interfaces. For example, calendars and video editors serve different purposes, but the linear structure of their timelines suggest an analogy. This structure is used to define the main objects of interest (events/video-clips), their presentation (vertical/horizontal layout), and their behavior (moving events/video-clips along the timeline) (Figure 1).

Belidor's use of structure as a first-class concept makes it...

(1) ...*parsimonious*. Belidor views UIs as relations between sets of objects and structures. The relational model permeates across an interface's conceptual model, its presentation, and behavior. For example, an interface's presentation is expressed as relations between marks (presentation objects) and encodings (presentation structures). Behaviors simply describe updates to these relations. Belidor's parsimony lets us describe these different phenomena in similar terms.
(2) ...*expressive*. Belidor can describe a wide variety of interactive systems, ranging from simpler interfaces like text-messaging (Figure 2), to complex professional tools like video editors (Figure 1). It can also describe non-graphical interfaces like screen readers (Figure 10) and hardware devices (Figure 11). Section 4 provides a gallery of examples that explain how Belidor's language features contribute to its expressivity.

Finally, we demonstrate that Belidor serves as an effective representation for surfacing analogies (a) by using a simple off-the shelf algorithm to automatically make analogies between compiled Belidor specifications (Section 5.1) and (b) with three case studies that show how Belidor reveals analogies, helps transfer ideas between user interfaces, and describe design patterns as analogies (Section 5.1). The resulting analogies are presented with an interactive analogy viewer.[3] In summary, this paper contributes:

(1) Belidor, a specification language that foregrounds the structure underlying interactive systems (Section 3).
(2) An example gallery demonstrating Belidor's expressivity by describing interfaces ranging from simple to complex GUIs (eg. color pickers to video editors), and other modalities like screen readers and tangible devices (Section 4).
(3) Three case studies of analogous interactive systems, demonstrating how Belidor serves as a fertile ground for finding and representing analogies (Section 5).
(4) An off-the-shelf pattern matching algorithm adapted to find analogies on 45 pairs of Belidor specifications (Section 5.1).

By seeking to make analogies between user interfaces, we see Belidor as a step towards a language that helps designers reason about the design space of interactive systems. In Section 6, we discuss future work through the lens of three scenarios: (a) using Belidor specifications to find insightful analogies during the interface design process, (b) using Belidor's semantics as the basis for a UI implementation framework, and (c) helping designers make *structural* design moves by making changes in underlying structure more explicit. We envision that Belidor's notion of *structures* is central to interface design practice, and that a future iteration of Belidor can make those design choices more explicit.

2 Related Work

2.1 Analogies and Structure Mapping Theory

Analogical reasoning has long been studied for problem solving. Analogies are used for math problems [52], programming [14], science [19] and design [13]. Analogies make important aspects of the problem domain more salient, leading to solutions. However, their effectiveness is modulated by expertise [13], how the analogy is presented [27] or recalled [43], and cognitive constraints like limited time or increased mental load [28, 63].

[1] The GoG was the blueprint for popular tools like ggplot [64] and Tableau [61].
[2] Bernard Forest de Bélidor published the first known book to use diagrammatic arrows in engineering diagrams in 1753.
[3] Analogy Viewer: https://belidor-specification.github.io/

Gentner [22] developed the Structure Mapping Theory (SMT) as a computational model of analogy. SMT posits that people prefer distant analogies that invoke the *relationships* between objects from one domain to another, rather than mapping the properties of the objects alone. For example, to make an analogy between a battery and a reservoir, the relations between objects (eg. reservoir/batteries are *filled with* water/energy) are more important than attributes (eg. reservoir/batteries are wet/dry). In particular, analogies that map higher-order relations—relations of relations like causal relations—are more novel and generative, historically being the source of scientific advances (eg. Rutherford's orbital model of the atom supplanting the plum-pudding model) [26]. The analogies people produce depend on a variety of factors. For example, a person's goal in problem solving influences the domains they recall, changing the input to the structure-mapping process and therefore the output analogy [23]. Furthermore, analogies are influenced by the salient aspects of the source domain. For example, an asymmetric metaphor like "the butcher is like a surgeon" (methodical) has a different meaning than "the surgeon is like a butcher" (messy) because each profession makes different connotations more salient [24]. Therefore, a person's goals and conceptualization of a domain can influence they analogies they produce.

SMT suggests that a representation that denotes how objects relate can surface analogies. For interactive systems, objects include presentation objects like individual UI elements (eg. buttons), as well as conceptual objects like conversation in a messaging application. Belidor's key insight is that objects of a certain type often relate to one another with simple *structures*.[4] This applies to conceptual objects, like a linear timeline affecting text-messages, as well as visual objects, like a vertical stack of rectangles, and actions like dragging a canvas object in two-dimensions. By making *structures* a first-class concept, Belidor facilitates their direct comparison across UIs. Furthermore, mapping structures to other structures, such as when defining a visual encoding, constitutes a high-order relation. Then, two applications that "use" conceptual structures in similar ways constitute a stronger analogy. For example, a calendar and a video editor both have a linear conceptual structure for their timeline, and both map it to a linear visual structure (vertical/horizontal). In sum, Belidor was designed to facilitate the kinds of comparisons that the SMT predicts help make distant analogies.

2.2 Analogical Transfer in HCI

Early developments of GUIs used metaphors to make interfaces more relatable [8, 34]. This became the basis for common metaphors like the desktop metaphor, menus, and buttons [2, 46]. While these real-world metaphors have proven their use (though not without criticism [32]), we focus on analogies between user interfaces.

2.2.1 Analogy from Visual Representations. Researchers have used machine learning techniques on visual representations of user interfaces to computationally produce analogies — often with screenshots, but also sketches [33]. This problem is often framed as finding or generating similar user interfaces for inspiration [4, 11, 33, 50]. For example, Lu et al. [44] presented Misty, a node-and-wire interface to operationalize conceptual blending, a form of analogy [62]. Users can combine application screenshots, which are fed to a multi-modal language model that generates front-end code. While these black-box systems can produce impressive outputs, they provide little insight on why an analogy holds, limiting the ideas they can generate. Researchers have also used other machine learning techniques to infer UI semantics from screenshots [38, 60, 66, 67]. However, these approaches primarily give meaning to individual UI elements. For instance, Jiang et al. [38] uses the Enrico and VINS datasets of mobile applications which only categorizes UI elements like buttons, text input fields and date pickers [11, 42]. Wu et al. [66] used an unsupervised approach to refine these labels with semantic information to match UI elements across application screens, for example identifying a text-field used for login credentials. Nonetheless, these UI element categories scarcely cover the breadth of the user interface design space and fail to adequately express many concepts in creation software, like the video editor's timeline. Furthermore, identifying UI element semantics does not amount to inferring the application's semantics as a whole system. For example, the same video clip in a video editor can be presented in multiple ways: as a thumbnail in the media pool, and as a resize-able object on a timeline. The subtle relationships between these views have yet to be captured by screenshot understanding techniques.

To avoid the trappings of visual representations, Belidor explicitly denotes deeper UI semantics than what can be gleaned from visual appearances, making distant analogies more accessible.

2.2.2 DOM-Retargeting. One approach to operationalize analogies on the web is to map parts of one website's Document Object Model (DOM) to another website, known as *DOM-retargeting* [5, 6, 41]. For instance, Kumar et al. [41] presented Bricolage to transfer the visual style between websites. The authors argue that HTML makes developing a consistent model between websites difficult and therefore train a neural network on human-produced comparisons of webpages. This suggests that the semantic information necessary for style transfer was either missing or diluted in the DOM. To explain this, Benson and Karger [5] contend that modern HTML fails to adequately separate the UI's presentation from its content. For example, today Slack's webapp uses upwards of 5 nested layers of divs for a single message, making it unclear which div should stand for the message. The authors present Cascading Tree Sheets (CTS) to specify layout with CSS-like rules that graft parts of a content-HTML document onto a layout-HTML file, or the inverse. This approach enables a variety of use-cases, including DOM-retargeting. However, the authors acknowledge that CTS only allows retargeting if both websites use the same content structure. Even then, HTML is best suited for representing hierarchical, document-like content. For example, an event in Google Calendar is positioned using a pixel-level calculation in Javascript: this means that its conceptual meaning as a range of time cannot be drawn from the DOM.

In contrast, Belidor directly represents interface semantics with coarser, structural relations. This holistic approach limits Belidor as an implementation tool, but frees it to express a broader range of interactive systems with a consistent and comparable language.

[4] We should however disambiguate between Belidor's notion of *structure*, and the "structure" in Structure Mapping Theory. Belidor's *structures* are a specific of kind relation inspired by data structures from computer science. Meanwhile, SMT refers to the "relational structure", the organization of relations as a whole. In other words, Belidor *structure* is only one relation, among others, in SMT's Relational Structure.

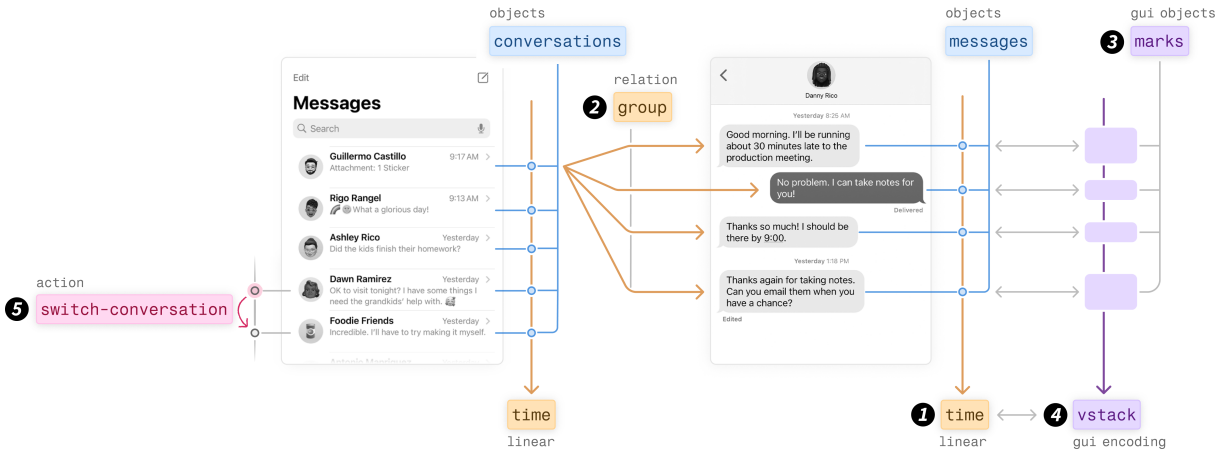


Figure 2: An overview of a simple messaging application (iOS Messages). The linear structure of time affects conversations and messages (❶). Conversations group (map to many) messages (❷). To present messages to the user, they are mapped to GUI marks (❸) as a vertical stack (❹) according to the temporal structure. Changing the selected conversation occurs along the temporal structure (❺).

2.3 Representing Interactive Systems

2.3.1 MBUID. The 1990s saw a flurry of research around *Model Based User Interface Design* (MBUID) [39, 54, 55, 59]. With this approach, designers modeled the problem domain with tasks that user could carry out, which could then be translated into user interface actions and elements, such as filling out text boxes, selecting drop-down menus, and clicking buttons using a UI semantics like Abstract Interaction Objects [18]. For example, Nichols et al. [51] generated remote control user interfaces, with the additional constraint that the resulting interface should match those that the user is familiar with. The first key limitation of MBUID was that task models were too constraining to describe many use cases, such as creativity support tools. Furthermore, the UI semantics were limited to simple components like buttons, sliders and text boxes. As a result, MBUID were best suited for developing complex forms, which by 2010 were already relatively easy to program. In contrast, Belidor can express the much broader range of user interfaces we use today, including complex creativity support tools.

2.3.2 Concept Design. More recently Jackson proposed *Concept Design* (CD), a theoretical framework that describes the concepts underlying user interfaces [37]. Concepts are units of functionality that the user must understand to use an interface. Examples include the social media up-vote, the file-system folder, and text-editing styles. Concepts can be represented with an Alloy specification of their state machine, alongside a natural language description of their purpose. However, Concept Design does not describe how concepts are presented to the user. This helps design clean concepts but prevents the comparison of presentation. However, an interface's presentation is important to its complete description. Belidor can make analogies that consider the conceptual and presentation layers, as well as their relationship.

2.3.3 Interaction Substrates. Mackay and Beaudouin-Lafon define Interaction Substrates as containers that provide structure and constraints to their objects of interests [45], and support dependencies with other substrates. For example, an interactive music sheet is a substrate for musical notes which can constrain notes to lie within a scale. It maintains a dependency with the *display substrate*, a two-dimensional array of pixels, for rendering. Mackay and Beaudouin-Lafon argue that the structure that underlie objects of interest gives them meaning (eg. music sheets communicates a musical context) and suggests affordances (eg. moving notes up and down in pitch), and are therefore important for designers to consider. They provide generative principles for substrates to help designers make more powerful and simple user interfaces.

Like Interaction Substrates, Belidor recognizes structure as a central to user interfaces and hence treats it as a first-class concept. However, while Substrates bundle structure and constraints, Belidor treats them separately. Furthermore, the dependency between *structural substrates* and the *display substrate* is similar to Belidor's mapping between the conceptual and presentation layers.

2.3.4 Model-View-Controller. Like interface designers, programmers are keenly interested in understanding user interfaces, or at least how they are built. UI toolkits like React, Swift, and Qt must be able to specify the application's conceptual model, its user-facing view, and how the user controls the model. Krasner and Pope [40] first argued for the separation of these three concerns (Model View and Controller). Since then, UI frameworks have organized these layers in different ways to improve developer experience. For instance, React encourages separating the concerns on a per-component basis, rather than for the entire application [36].

Belidor describes these layers with the same language of objects, structures and relations. Therefore, analogies can be drawn by relating part of one layer onto another layer. For example, that both calendars and video editors use a timeline structure is interesting, but the analogy especially compelling because they both use this conceptual structure for presentation (vertical/horizontal) and behavior (dragging events/video-clips along the timeline).

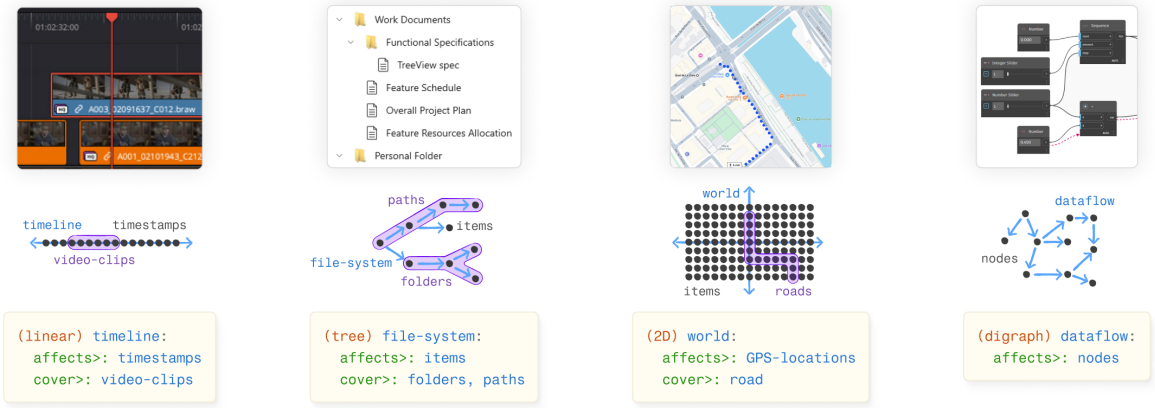


Figure 3: Examples of conceptual structures (blue arrows) and covers (purple highlight) in four different applications.

3 Belidor: A Relational Model of Interactive Systems

Belidor is an entity-relationship model for interactive systems. It is embedded in YAML, whose key-value model is familiar and well suited for denoting relations. This section explains Belidor's syntax and semantics, along the with rationale for its language features.

3.1 A Messaging App through Belidor's Lens

To build intuition for Belidor's semantics, we begin with an overview analysis of a simple messaging application (Figure 2). Belidor builds on a relational model of interfaces: we begin by asking "what are the objects of interest, and how do they relate"? In a messaging application, messages are central objects that relate to one another temporally. That is, a linear structure (of time) *affects* these messages (Figure 2 ❶). Messages also relate to one-another based on the conversations they were sent in. Thus, conversations themselves are objects which *group* messages (Figure 2 ❷). A messaging application typically only shows a *subset* of its messages at a time, mainly the messages in the selected conversation.

Graphical user interfaces have their own objects and structures. Namely, GUIs have *mark* objects (e.g. text boxes) and *encoding* structures (e.g. a vertical stack). To present information, the objects and structures of the conceptual model are mapped to the GUI presentation. In this example, we map messages from the selected conversation to text marks (Figure 2 ❸), and time to the vertical stack (Figure 2 ❹). Similarly, the conversations view vertically stacks conversations using time (Figure 2 left).

Finally, Belidor describes how actions create, update, and delete existing objects and relations. For instance, switching conversations is a *navigation action* that updates the selection of conversation by moving it along the temporal structure (Figure 2 ❺).

The next sections explain these concepts with Belidor's syntax.

3.2 Structure & the Affects Relation

Objects of a common type are often organized by an underlying *structure*: messages are organized linearly over time, files in a file system are organized in a hierarchy, and shapes on a canvas are organized in a two-dimensional space. Belidor reifies structure as a first-class language construct. For example, to denote that a linear structure of time applies to messages, we write:

```
(linear) time:
  affects>: messages
```

The `affects>` relation indicates that the source structure applies to the target(s). Concepts are declared when they are used. Here, the statement declares: 1) `time` is of type `(linear)`, 2) `messages` is a set of objects, and 3) a relation where `time` affects `messages`.

In Belidor, objects (eg. `messages`) are sets. A structure affecting a set of objects denotes a directed graph, where individual objects are nodes (eg. individual messages), and the structure supplies the directed edges. Belidor's standard library provides five structure types based on their shape: `(linear)`, `(2D)`, `(3D)`, `(tree)`, and `(digraph)` (Figure 3). We find these are sufficient for many interactive systems, but authors can also define their own (Section 3.4).

Rationale. Belidor structures reify a ubiquitous kind of relationship between interface objects. SMT predicts that mapping relationships like these structures leads to analogies between user interfaces (Section 2.1). Furthermore, reified structures can be the source and target of relations (Figure 4). This helps the language be composable because structures can be used and related in open-ended ways. In SMT, such "relations of relations" are called "high-order relations" and are especially important for distant analogies.

3.3 Subsets & Queries

When user interfaces present or afford actions on objects of interest, they often focus on a particular subset at a time (eg. selected items). *Subsets* can be declared with a `subset>` relation, or more concisely in one place with a dot syntax: `conversations.selected`. *Queries* can reach across subsets with the arrow syntax (`->`) to carry out a proper function. For example, `conversations.selected->messages` returns *the set of messages in the selected conversation.* The dot operator takes precedence over the arrow, meaning that they can be chained into more complex queries (Section 4.2).

Rationale. Queries are used extensively in the presentation (Section 3.6) and behavior layers (Section 3.7). The dot and arrow syntax makes these queries concise, and lets them be the target of relations.

As a result, relations can apply to specific subsets without requiring many new identifiers (eg. for each type of selection).

3.4 Groups

Belidor denotes one-to-many relations with the `group>` relation. For example, a conversation maps to many messages.

```
- conversations:
    group>: messages
```

Composite groups. We often want objects to encapsulate other objects and relations. For example, each person could have their own name and phone number. Belidor denotes this relation as a `group foreach>`, where group items do not overlap:

```
- people:
    group foreach>:
        - /name
        - /phone-number
        - (gui) /contact-info-view
```

Note that prefixing an attribute with `/` functions as namespacing. To access the attribute outside of the `group foreach>` clause, it is prefixed with the group's name eg. `people/name`. The right-hand-side of the `group foreach>` relation can be a complete specification with structures, presentations and behavior. This allows us to refactor an individual component specification as a collection (e.g. changing a single web page specification to a collection of web pages).

Named groups. Groups can be given a name for reuse, turning it into a named group. For example, structure types (eg. linear, 2D, tree) are simply named groups with attributes.

```
- def (tree):
    group foreach>:
       - (linear) /depth-first-ordering
       - /branches
- (tree) reply-structure:
    affects>: messages
    cover>: threads
    /branches =: threads
```

Structures, like other types, can have attributes. For example, trees have attributes like depth-first and breadth-first ordering, roots, leaves and internal nodes. In the messaging application, threads are branches of the reply structure's tree. When instantiating the reply-structure with the tree type, its attributes can be aliased (`=:`) to objects and structures. Aliasing two sets means they are the same; the Belidor compiler will combine them such that either name can be used interchangeably throughout the specification. When used in type instantiation, it binds external concepts (eg. `threads`) to internal attributes (eg. `/branches`).

Rationale. Named groups are Belidor's primary abstraction mechanism, used to bundle information into one place. This is useful for Belidor's core constructs defined in the standard library, such as structure types, view types, and actions. Authors can also use named groups to define reusable components with their own internal conceptual, presentation and behavior. Using a single syntactic construct helps harmonize the language and maintain parsimony. Since named groups function as types, we will use those words interchangeably throughout the paper.

3.5 Covers

A cover is a group whose elements are contiguously connected by a structure. For example, selecting text creates a ranged selection which acts as a group on the characters (the selection maps to many characters). Critically, these characters are strung together by the text buffer's linear order.

```
- (linear) text-buffer:
    affects>: characters
    cover>: range-selection
```

Here, `cover>` denotes that `range-selection` covers characters along the buffer. Figure 3 illustrates more covers.

Rationale. By combining the notions of groups and structures, covers describe a surprising range of UI phenomena. In an implementation, covers over different structures would be programmed differently: linear structure covers are defined by a start and an end point, two-dimensional structure covers are regions of space, covers for tree structures are sub-trees, etc. These technical details are abstracted away and unified as the same concept.

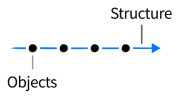
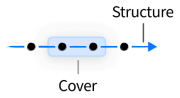
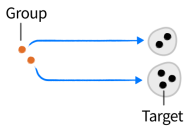
This uniform representation results in useful analogies. For example, ranged selections can be used in different structures: selecting a range of a timeline on a calendar is analogous to a ranged marquee selection on a canvas. Furthermore, the analogy from Masson et al.'s Textoshop [48] between a text editor and an image editor relied on "words as pixels, sentences as regions." In the text editor, sentences cover words along the linear structure of text; in the image editor, regions cover pixels in the two dimensional canvas structure. While the structures and objects of interest are different, Textoshop's analogy rests on mapping one cover to another.

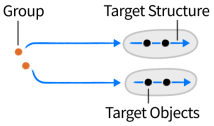
3.6 Presentation Specification

User interfaces present parts of the conceptual model to their users. Belidor's standard library defines presentations with *marks* (objects) and *encodings* (structures). Common view types such as GUIs and screen readers extend this definition with their own subsets of marks and encodings. For example, `(gui)` is a type with visual marks (eg. text and shapes) that can be laid out with visual encodings (eg. clustering and stacking). In the messaging application, messages (from the selected conversation) map to the view's text marks, and the time structure maps to the vertical stack encoding. Note that the `<>:` operator is shorthand for a map relation.

```
- (gui) messages-view:
    /encodings.vstack <>: time
    /marks.text <>:
        conversations.selected->messages
```

Rationale. Marks and encodings naturally fit Belidor's objects and structures, and mapping the conceptual model to the presentation is a natural extension of Belidor's relational semantics. This demonstrates Belidor's parsimony: the presentation semantics simply build on existing language constructs. This structural perspective on UI presentation approach has already found success in information visualization (e.g. Bluefish and GoFish [56, 57]) and

Relation	Meaning	Example Syntax	Example Schematic
Affects `affects>`	Map objects onto a structure.	`(linear) time:` — Structure `affects>: messages` — Objects	
Cover `cover>`	Map objects onto contiguous objects in a structure.	`(linear) text-buffer:` — Structure `cover>: range-selection` — Cover Objects `affects>: characters`	
Group `group>`	Source objects map to many target objects, acting as a group.	`conversations:` — Source Objects `group>: messages` — Target Objects	

Composite Group
`group foreach>`
	Encapsulates a specification for each element of a group.	`videos:` — Group Objects
		`group foreach>:`
		`- /images`
		`- (linear) /time` — Target Specification
			

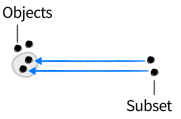
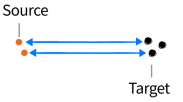
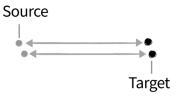
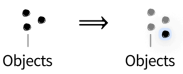
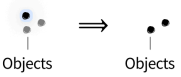
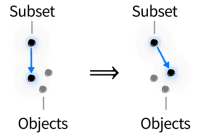
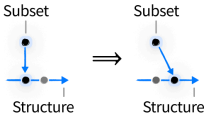
Subset `subset>`, `a.b`	Define the target object as a subset of the source object.	`files:` — Source Objects `subset>: files.selected` — Target Objects	
Map `mapto>`, `a → b`, `a <>: b`	Map source objects to target objects.	Source Objects `files→file-paths` — Target Objects	
Alias `alias>`, `a =: b`	Equate the source and the target. Both names become equivalent.	Source `file-system/leaves := files` — Target	
Create `create>`	Add to a set of objects.	`(action) new-file:` — Action `create>: files` — Objects	
Delete `delete>`	Remove from a set of objects.	`(action) delete-file:` — Action `delete>: files.selected` — Objects	
Update `update>`	Update the contents of a set or the instance of the relation.	`(action) select-item:` — Action `update>: items.selected` `subset> items` — Relation	
Directional Update `directions>`	Map structure onto action.	`(action) move-shape:` — Action `update>: canvas cover> shapes` `directions>: canvas` — Structure	

Figure 4: Belidor relations and their syntax. The schematics illustrate how a relation influences an identifier's concrete elements. For example, the first row shows how individual objects (messages) are strung together by the time structure (arrow).

in interaction design (e.g. Interaction Substrates, where structural substrates map to display substrates [45]). Moreover, Belidor's presentations are sufficiently expressive to describe various modalities. Section 4.3 explains how to specify interactive systems beyond GUIs such as screen readers and tangible hardware devices.

3.7 Behavior Specification

To describe system behavior, Belidor specifications list the *actions* a user can take. Actions can *create*, *update*, and *delete* existing objects and relations. For example, a messaging application lets users send and delete existing messages by editing the set of messages:

```
- (action) send-message:
    create>: conversations.selected->messages
- (action) delete-message:
    delete>: messages.selected
```

The user can switch between conversations by updating the subset relation between `conversations.selected` and `conversations`, changing which conversation is selected:

```
- (action) select-conversation:
    update>: conversations.selected subset>
        conversations
```

Actions with Structure. Structures can influence actions. For example, a *linear* set of actions suggest that the user has two choices — forward and backwards. Structured actions are commonly used to specify navigation interactions. For example, a keyboard user can navigate between conversations using their keyboard. Pressing up and down lets the user move between conversations in the order they are presented, their temporal ordering.

```
- (action) navigate-conversations:
    update>: conversations.selected subset>
        conversations
    directions>: time
```

Note that `directions>` is like `affects>` for actions. Belidor distinguishes them syntactically to keep the structure under the action specification, which we found easier to read and write.

Seen in this way, navigation covers interactions ranging from changing a selection, to directly manipulating objects and navigating between screens. Navigation actions also help make analogies: for example, navigating between selected conversations is similar to navigating between files in a file browser.

Presenting Actions. Actions, like other objects, can be presented in views. In a messaging application, the user selects conversations in the conversation view.

```
- (gui) conversation-view:
    /marks.text <>: conversations
    /marks.hitboxes <>:
      - (action) select-conversation:
          update>: conversations.selected subset>
              conversations
```

In GUIs, a mouse cursor performs a hit test and triggers the actions mapped to the mark, which is expressed in the standard library with the query `gui/cursor->gui/marks.hitboxes->action`. This approach to expressing interaction techniques is adapted from Conversy's Picking Views framework, which can describe interaction techniques ranging from buttons and scroll bars, to snapping objects to magnetic lines on a canvas [16].

Rationale. Interaction is a central aspect of user interface; Belidor could hardly describe which UI features are analogous without semantics for behavior. To do so, Belidor has semantics for editing sets of objects and relations based on the Create-Read-Update-Delete (CRUD) software architecture. While action semantics are new from the conceptual and presentation layers, they nonetheless naturally fit with Belidor's relational semantics.

Moreover, the behavior specification discriminates static parts of the conceptual model from those the user can interact with. This is important for concepts like selection: in the conceptual model, a selection is just a subset, but the behavior specification highlights that it is the funnel through which interactions take place. This also helps analogies: the relational pattern surrounding a selection is easily identified and mapped between UIs that use selections.

3.8 Design Reflection

Belidor was the result of many iterations. In this section we reflect on the design choices that have led to Belidor.

3.8.1 Facilitating Analogies. Surfacing analogies is one of the main pressures on Belidor's design. To that end, we drew from Structure Mapping Theory (SMT; see Section 2.1), which posits that analogies depend on the relations between objects and relations in the domain. As explained in Section 2.1, this perspective naturally led us to the entity-relationship model. We also promoted structures as first-class concepts, which lets us specify relations between structures (eg. between conceptual, presentation and behavior specifications). These are high-order relations, which SMT argues make for more distant and novel analogies.

Facilitating analogies also directed lower-level design decisions. Belidor needs to adequately discriminate between UI concepts to suggest meaningful analogies. Otherwise, unrelated ideas can unhelpfully be placed in an analogy. For example, Section 3.7 discussed how behavior specifications separate selection subsets from other subsets. Similarly, subsets and mapto relations can both be formalized as proper mathematical functions. However, their meaning was too different to conceptually merge them without engendering unhelpful analogies. We achieved Belidor's final balance by iteratively checking against example analogies.

3.8.2 Incremental Specification and Belidor's Usability. We now systematically evaluate Belidor's usability with Green et al.'s Cognitive Dimensions of Notation [30].

Premature Commitment forces authors to make decisions before having the necessary information. A central goal for Belidor was to let authors only specify the aspects of the interface they care about without having to describe "the rest of the world". This is especially important because interactive systems are overwhelmingly complex and designers are only ever interested in a few features at a time. Belidor achieves this goal with "incremental specification", where authors gradually build up a specification one relation at a time.

Incremental specification manifests in the design of the language:

(1) The order of the declarations does not matter, since every statement adds to a collection of relations.
(2) Objects are defined when they are used, and relations can be defined in place with Belidor's nested infix relation syntax. For example, `select-conversation` in Section 3.7 is defined whilst being mapped to `hitboxes`.
(3) Belidor does not force the separation of model, presentation, and behavior. An author can analyze a user interface by studying its presentation (eg. a screenshot) and slowly uncovering the conceptual model through it. As they write the view specification, the objects and structures that marks and encodings map to are defined in place. Nonetheless, the model–presentation–behavior layers easily can be separated via simple refactoring if the author so chooses.

Incremental specification lets authors begin where they want with few downstream costs, reducing *premature commitment*. However, this flexibility implies that authors might produce different specifications for the same interactive system depending on what they focus on. While simply changing the order of declarations will compile to the same representation, focusing on different features will not. This means the analogies drawn from these specifications will also be more specific to the designer's interests. This is consistent with how people make analogies in general: their goal influences how they conceptualize the source domain, which will in turn influence which aspect of the domain they will recruit in their analogy [23, 24] (see Section 2.1).

Abstraction Gradient is the mechanism by which a notation treats a a group of elements as a single entity. Named groups are Belidor's only mechanism for abstraction. When used for functional elements of an interface, Named Groups can be used as components in frameworks like React [36]. Since Belidor lacks for-loops and functional maps, named groups are also a convenient mechanism to denote repeated interface elements, forcing the author to view repeated items as components. In Section 5.4 we demonstrate that Belidor allows for abstractions *across* specifications through analogy: an abstract design pattern can be seen as a specification that is analogous to the subset of UI that use the design pattern.

Viscosity describes the notation's resistance to change. Belidor's incremental specification significantly lowers the viscosity of adding and editing relations, since authors can do so anywhere. However, this flexibility comes at a cost: if the author changes their mind and decides to completely remove a relation, they must find and remove all of its uses. For example, a subset selection might be declared with the `subset>` keyword in one place and used elsewhere to define a selection action. However, this is only a problem when deleting a relation, which we found to be relatively rare. Overall, we deem the viscosity tradeoff worthwhile.

This issue illustrates a *hidden dependency*: an identifier does not show which items it relates to whenever it is used. For example, after defining `(linear) time affects>: messages`, every other use of `messages` does not show its relationship with `time`. However, this information is easily recovered from the compiled specification, such that a specialized editor could show all relations that apply to a selected identifier.

Secondary Notation carries informal information around the notation. This paper does not contribute a specialized editor for Belidor, nor a secondary notation for authoring specifications. However, many of our figures extensively annotate user interface screenshots to explain specifications. This suggests that an interface designer could analyze a user interface with Belidor's semantics by annotating UI screenshots. Belidor's semantics can therefore serve as the basis of a *Professional Vision*, a way of "seeing" and interpreting professional artifacts in the wild [3, 29]. Across disciplines, Goodwin's work on Professional Vision shows that annotating artifacts in terms of a conceptual tool, like Belidor's semantics, helps experts communicate in a common language while staying grounded in discussing concrete artifacts.

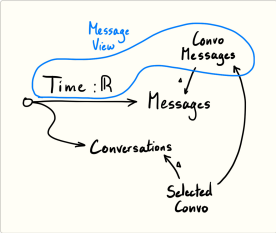
3.8.3 Limiting Descriptions of Interaction. With relatively few language primitives, Belidor expresses a wide range of interaction techniques but not without tradeoffs. First, Belidor currently does not bind physical inputs to actions. For example, Belidor does not specify how keyboard shortcuts map to action. For example, while Belidor describes that moving an object on an infinite canvas has a two-dimensional structure, it does not specify whether the drag is initiated by a mouse or touch input. However, input bindings invite significant complexity in the form of modes. For example, a Bezier tool in a vector drawing application changes its functionality depending on which keyboard modifier keys are pressed, resulting in complex state machines. Moreover, the added complexity does not pay off because which input device triggers an action, and the condition under which it does, does not help make analogies. By denoting the structure of an action (eg. for direct manipulation), Belidor already captures much of what makes an input device unique. In particular, Card et al.'s design space of input devices largely revolve around the input's structure (eg. 2D mouse). As a result, we decided against including syntax for input-bindings.

Another key limitation is that Belidor only seeks to describe what is changed, without saying how it is changed. In other words, it does not describe computation. As a result, a selection action does not actually describe that the clicked object becomes selected; only that the selection changes. However, adding notation for arbitrary pseudo-code to describe computations would add significant complexity to the language. Furthermore, code is too flexible to lend itself to comparison; since there are many ways of achieving the same outcome, comparing code snippets is difficult. Therefore, given the limited benefits and substantial added complexity, Belidor currently omits denoting any specific computation.

3.8.4 Different Syntactic Paradigms Influenced Belidor's Semantics. Belidor's design took many twists and turns. While the insight to reify structure came early, we experimented with different syntactic paradigms to find how to best express the language's semantics. In this process, each paradigm left a mark on Belidor's semantics.

Inspired by the UML diagrams [10] software engineers use to discuss architectural decisions, we began with a diagram language for whiteboards (Figure 5 ❶). The mechanical tedium of writing and erasing on a whiteboard pressured the language to be concise and minimize refactoring. This was the source of Belidor's incremental specification which survived the following iteration (Section 3.8.2). However, the whiteboard's affordances required us to solve many problems simultaneously: designing useful semantics, encoding

❶ Diagrammatic



❷ Prolog

```
linear(time)
time structures messages
time structures conversations
conversations mapto_many conversations
selected_convo subset conversations
selected_convo mapto messages
convo_messages subset messages

gui(message_view)
message_view represents
   convo_messages as textbox
   with time as vertical_stack
```

❸ Earlier YAML

```
objects:
  - name: messages
    subsets:
      - convo_messages
  - name: conversations
    subsets:
      - selected_convo
    groups:
      - messages

structures:
  - name: time
    type: linear
    sets:
      - messages
      - conversations

views:
  - name: messages-view
    type: gui
    objects:
      - messages
    structures:
      - time
```

❹ Belidor

```
- (linear) time:
    affects>: messages, conversations

- conversations:
    group>: messages

- (gui) messages-view:
    /marks <>:
      conversations.selected→messages
    /encoding.vstack <>: time
```

Figure 5: Belidor's predecessors. Actions are omitted for brevity. ❶ An entity-relation diagram. Arrows labeled with a triangle are subsets, the unlabeled arrow is a mapto relation. The circle arrows are structures; time is typed as 'linear' using the ℝ symbol. ❷ Relations are defined as infix operators in Prolog. ❸ Early version embedded in YAML using standard key-value notation. Items are defined together based on their kind (object, structure, view, action). ❹ Belidor, as described in this paper.

them graphically with ergonomic and legible symbols, minimizing layout problems, etc. While ultimately we believe a diagrammatic version of Belidor may help designers, we decided to pursue text-based notation to focus on defining effective semantics.

We next embedded Belidor in Prolog [21] which forced us to more formally define Belidor's relations (Figure 5 ❷). For example, we found that presentations could also be presented relationally. With consistent semantics, we implemented logical rules to make inferences about specifications using Prolog's engine. For example, we could find every object affected by a linear structure. While exciting, these inferences were not particularly useful. Syntactically, Prolog discourages nesting, resulting in uniform specifications that were easy to write but difficult to read.

This insight led us to explore key-value object languages (Figure 5 ❸). Early versions in YAML followed standard key-value semantics. This was verbose and diffused concepts across the specification, meaning small changes required edits across the file. To make the language more concise we found we could 1) reorganize relation definitions by placing the source as the key to a nested dictionary, whose keys and values are respectively relations and targets, and 2) replace the sprawl of definitions like `conversation_messages` with the query syntax. This reduced *viscosity* [30], making edits easier. We also developed the named group concept to unify structure and view type definitions, making the language significantly more parsimonious (Figure 5 ❹).

3.9 Summary

Belidor is a declarative specification language for interactive systems. In particular, it highlights the *structure* underlying the conceptual, presentation and behavior aspects of a user interface. Using the same language features across layers makes Belidor *parsimonious*. For example, a visual encoding may seem unrelated to navigation, but Belidor describes both in terms of structure. Similarly, Belidor uses queries both to describe how a view only presents some information, and how an action only applies to selected objects.

The next section demonstrates Belidor's expressivity, in spite of its parsimony, with a gallery of examples. Section 5 then shows how Belidor surfaces analogies between user interfaces with case studies and a simple analogy algorithm.

4 Example Gallery

This section demonstrates that Belidor's semantics are *expressive*, showing how they can concisely describe a wide range of interactive systems. The following examples demonstrate how user interfaces make use of their structures (Figure 6, Figure 7), how queries can specify complex presentations and behaviors (Figure 8, Figure 9), and presentation specifications for non-visual interfaces in different modalities (Figure 10, Figure 11).

4.1 Designing with Structure

Structures are the backbone of an application, finding their way into the presentation and behavior layers. The messaging example of section Section 3.1 primarily revolves around a linear structure for time, but interactive systems can use other structures too (Figure 3).

4.1.1 Color Picker. Colors are typically encoded as points in a three dimensional space. While the interface could display a three dimensional color cube, it is not a convenient way to pick colors. Instead, color pickers use structures derived from the color space. The standard library provides the `(3D)` structure with attributes for its axes (eg. `hsv-color-space/z-axis`) and cross cutting planes (eg. `hsv-color-space/xy-plane`). By describing the color space as an instance of the `(3D)` structure, we can use its attributes to describe an area picker for the saturation-value dimensions of the color space, and a slider for the hue dimension (Figure 6). Depicted like this, the design of the traditional color picker solves a structural problem — how can the color space be represented in a flat GUI? The solution is to break the `(3D)` space into component dimensions both visually (as `(2D)` and `(1D)` components) and behaviorally (through `(2D)` and `(1D)` navigation of a picked color). Structures, and the dependent structures they imply, can serve as resources that help designers explore alternatives.

4.1.2 Code Editors. Structures provide affordances for presentation and behavior. In code editors like VSCode, language servers provide additional structure to text files. Beyond the linear order of a simple text buffer, tokens (character covers) can be given more structures. For example, an abstract syntax tree (AST) lets the user collapse blocks of code (Figure 7). This is possible because `(tree)` provides an attribute for `/subtrees` covers. The code view then only shows a subset of the subtrees. The `toggle-code-block` action,

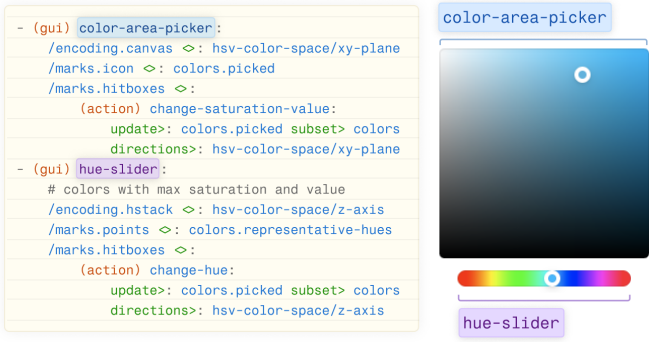


Figure 6: *Color picker*: `color-area-picker` describes the saturation-value area picker with the color space's `/xy-plane`. The `hue-slider` similarly uses the `/z-axis` of the color space. Each view presents actions to pick colors (`change-saturation-value` and `change-hue` respectively). Both result in the same update (changing the picked color), but use their respective structure as directions. This describes how the user directly manipulates `colors.picked` in each view.

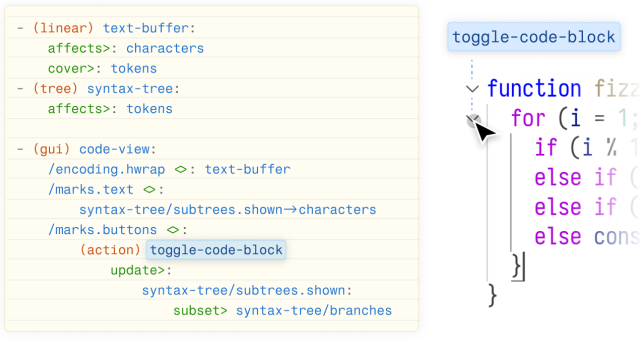


Figure 7: *Code Editor*: The abstract syntax tree enables more interactive views. The `code-view` only shows the open code blocks (`syntax-tree/subtrees.shown`). The view also presents the `toggle-code-block` action for collapsing subtrees, denoted as changing the set of shown subtrees.

which is presented in the view as buttons, can then change the set of shown subtrees, influencing the view. In summary, adding the AST structure opened possibilities for presentation and behavior.

4.2 Queries Enable Concise Presentation and Behavior Specifications

Section 3.1 detailed how Belidor's query syntax precisely expresses what views show and what actions change. We can take this concept further with concise specifications of sophisticated systems.

4.2.1 File Explorer. File explorers like macOS Finder and Windows File Explorer can show files in a "column view". This can be programmed by iterating across the folders in the selected file's path, producing and horizontally stacking views for each one. Belidor can express this in a single view without resorting to iteration (Figure 8). This is achieved with a single line query that finds the set of items (`items`) in the folders (`items.folders`) of the selected item's path (`items.selected->paths`). Together, the query becomes: `items.selected->paths->items.folders->items`. These items are encoded with a vertical stack and the columns emerge from a cluster encoding using the group formed by the folders along the path.

4.2.2 Spreadsheets. Queries can also precisely express exactly what an action will change. For example, a hallmark of spreadsheets is that changes propagate throughout the document (Figure 9). This is denoted with a single action that changes the selected cell's formula, its result, and the results of every cell that depends on the selected cell. This query is constructed with the `\descendants` attribute of the dependency graph, which finds every cell that depends on the edited cell. This example also demonstrates how a conceptual structure can afford targeting specific objects in an action. Note that this action does not explain how the computation is carried out, only what elements are influenced in the computation.

4.3 Beyond GUIs

When we think of user interfaces, graphical user interfaces typically first come to mind. They are indeed very common and perhaps constitute the bulk of professional and HCI interface design. However, over-indexing on GUIs can limit how we think about interactive systems in general. This section demonstrates how Belidor denotes non-graphical user interfaces on equal footing with GUIs.

4.3.1 Screen Readers. Screen-readers are an auditory presentation layer used by people with low or no vision. Figure 10 presents screen-reader views for a webpage. `(reader)` is an extension of presentations type just like `(gui)`, complete with mark and encoding attributes. The first (top) presents a naive screen reader specification. At first glance, since the reader view only has a linear encoding, it could present the webpage linearly. This can be accomplished using the DOM's `/depth-first-ordering` attribute defined in the standard library's `(tree)` type[5]. This specification highlights the most salient limitations of screen readers: without a wide range of encodings, like GUIs, designers must make non-linear structures fit into a linear encoding. In practice, designing a screen reader application is more about navigation than linearization. This is expressed with the `interactive-dom-reader` in Figure 10 (bottom). Its presentation is much simpler, and instead denotes navigating the DOM with the same `dom/depth-first-ordering` structure. In other words, the `static-dom-reader` (top) organizes content with its presentation layer, whereas the `interactive-dom-reader` (bottom) organizes its content with the behavior layer.

Trading off organization between layers is not unique to screen readers; it is also common with GUIs, for example when designing for the limited screen real estate of a phone. This forces designers to spread information across different views that the user can navigate between, just like `interactive-dom-reader`.

[5]We omit details about how divs contain text (also linear) as it is not necessary to this analysis. This is the benefit of incremental specifications in Belidor (Section 3.8.2).

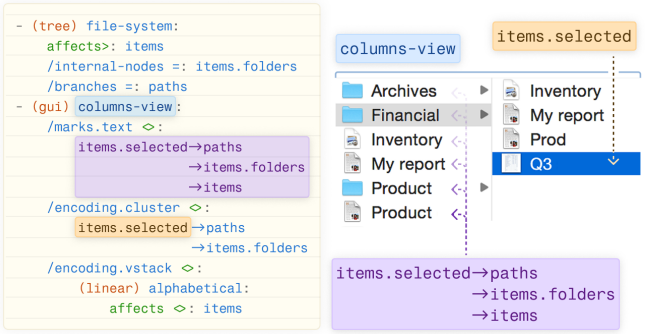


Figure 8: *A file system*: This view presents all of the items in the folders along the file's path. It uses a query that starts with selected items (`items.selected`) and finds the associated `/branches`, also named `paths` (`items.selected->paths`). Since branches cover the file-system tree, they act as a group on `items`. The query can then pick out folders along the path (`items.selected->paths->items.folders`), which are mapped to the cluster encoding. The standard library defines the tree's internal nodes (folders) as covers, meaning a query can peer into them to find the items themselves: `items.selected->paths->items.folders->items`.

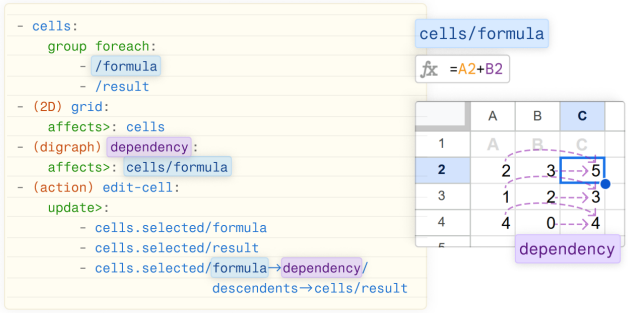


Figure 9: *A spreadsheet*: Editing a cell changes: the formula of the selected cell, its result, and the result of cells that depend on the selected cell. Cell dependency is modeled as a directed graph structure (`digraph`) over formulas. The `/descendant` attribute is a cover that describes elements downstream of each element, so a query can then find the descendants of the selected cell (`cells.selected/formula->dependency/descendents`). It then finds the associated result of those cells (`cells.selected/formula->dependency/descendents->cells/result`), describing the propagation of spreadsheet changes in a single line.

4.3.2 Elevator Interface. Belidor does not attempt to model the real world (eg. how a person's hand engages with buttons and knobs); but it can describe how the user interface models the world. There is no standard presentation type for hardware devices. Industrial designers have to define the affordances of their presentation layer and therefore define their own presentation type. For example, the author must define the elevator's interface as a new type (Figure 11) that can then be instantiated and mapped to the conceptual model.

Designing a hardware device involves designing the presentation layer's marks and encodings in concert with the conceptual model. Designers iteratively mold the presentation's type definition to a conceptual model, brainstorm what views it enables, which in turn influences the conceptual model. Belidor denotes this process by providing semantics for novel presentation types.

5 Operationalizing Analogies with Belidor

This section demonstrates how Belidor's representation provides fertile ground for making UI analogies manually and computationally. We present a simple computational method using an off-the-shelf graph edit distance algorithm to find analogies between pairs of specifications and discuss how the results can be interpreted. In three case studies, we demonstrate how Belidor can help highlight and explain UI analogies, in turn generating ideas making surprising connections across applications.

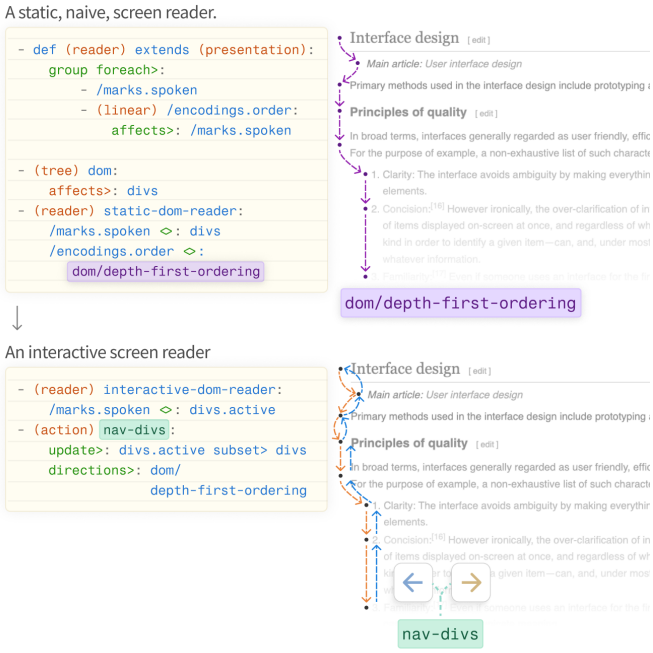
5.1 An Analogy Algorithm

> *"Solving a problem simply means representing it so as to make the solution transparent."* — Herbert Simon

Algorithms are generally easier to design when using an appropriate data structure for the problem. Prior work in HCI on style transfer between user interfaces shows that visual representations require sophisticated machine learning methods. To demonstrate that Belidor serves as a computationally friendly representation for operationalizing analogies, we present a simple algorithm that finds an analogy between pairs of Belidor specifications without datasets or expensive model training.

5.1.1 Algorithm Description. First, each specification is parsed and compiled into a directed graph representation of the specification: nodes are identifiers (objects, structures, and types) and arrows are labeled relations (Figure 4). Constructing an analogy is then equivalent to the *graph edit distance* problem (GED): which edges and vertices should be inserted or deleted from the first graph to produce the second graph? Unfortunately, GED is NP-Hard [68]. We use an off-the-shelf implementation of Abu-Aisheh et al.'s algorithm with the NetworkX package in Python [1, 31]. The algorithm casts the GED problem as path-finding by searching for the minimum cost edit path between the two graphs. Previous algorithms used A* to navigate the search tree, resulting in high memory demands. Abu-Aisheh et al.'s algorithm reduces memory complexity by using depth-first-search on a search tree ordered with Munkres' algorithm, and heuristically pruning sub-optimal branches.

To use the algorithm, we define cost functions for edit operations, which encode the matches we want to reward or avoid. For example, we strongly penalize matching nodes across conceptual, presentation and behavior layers (eg. matching `(gui)` with `(action)`) and edges labeled with different relations (eg. matching a `subset>` relation with `cover>`). Nodes with matching standard library types are also rewarded. This also rewards structures with matching shapes, which for example helps match linear structures with other linear structures. These costs were manually tuned, although not extensively. Note that this algorithm does not rely on the node names; it only uses the connectivity of the graph to encode the interactive



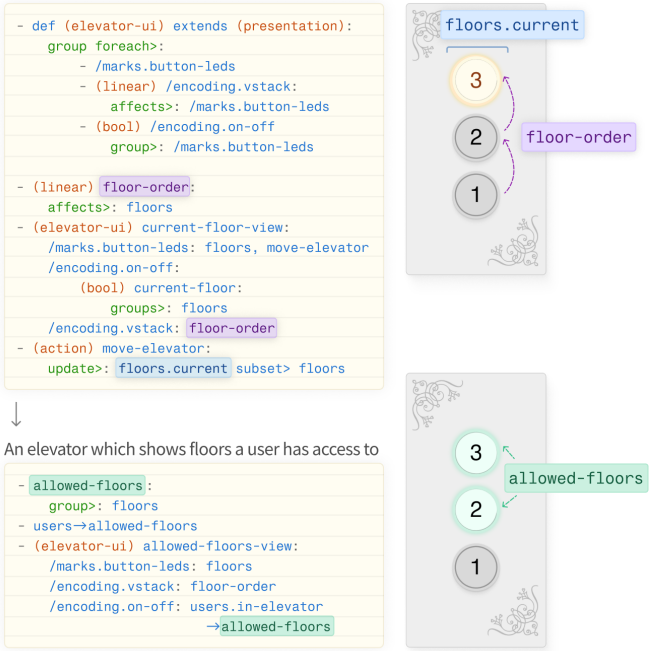


Figure 10: *A screen reader*: (Top) This webpage reader denotes how the view "linearizes" the DOM; presenting the user with a sequence of spoken divs. This sequence is derived from the depth-first-ordering attribute of the tree structure. (Bottom) In practice, screen-readers offload structural complexity onto the behavior layer. The interactive reader only presents one item at a time (`divs.active`), without an encoding. Instead, the DOM's depth-first-ordering is used for navigation action by changing which div is presented.

Figure 11: *An elevator interface*: (Top) The `(elevator-ui)` type defines its marks as button LEDs and its encoding as a vertical stack or as a boolean group, respectively representing the layout and state of the LED buttons. This is sufficient to express how the LEDs change as the elevator moves while also inviting different views. (Bottom) For example, an elevator in a corporate office could show which floors are available to the user. In this specification, users map to `allowed-floors`, which act as a group on floors. The view can then encode the LED state based on which floors are allowed to the user in the elevator.

system's semantics. To present the final analogy we prune isolated nodes, with no matching edge, from the mapping. Isolated nodes are meaningless to the analogy — since they are untethered, any isolated node match on one side can be matched with any isolated node on the other side of the analogy. In practice, the algorithm produces isolated nodes when it cannot find good edge matches to minimize deletion cost.

5.1.2 Automatically Generating Analogies. We applied the algorithm to 45 pairs of specifications derived from every possible pairing of 10 specifications—excluding pairing specifications with themselves. Results are mirrored for the 45 pairings (Figure 12).

Computation time grows quickly with large specifications. Therefore, we present the top scoring results after a maximum of two hours searching for analogies for each pair. The algorithm produces an iteration of results every time it finds a lower cost match. Pairings with less than four iterations are marked with an asterisk, as this suggests that the algorithm did not effectively explore the search space. The overlap of a pairing is scored by the proportion of edges in or at the boundary of the conceptual layer of the smaller specification found in the analogy. The algorithm minimizes the edit cost across all layers because the presentation and behavior layers help make appropriate matches by disambiguating conceptual nodes (eg. actions disambiguate selection from other subsets). However, to make sense of analogies we are primarily interested in conceptual overlap. These results are presented in Figure 12 and can be explored with the online viewer.[6]

The highest scoring pairing is between iMessage and Slack. This is not surprising because they belong in the same product category, and therefore presumably have a similar conceptual model. The computationally generated analogy confirms this, mapping messages and people across applications and matching iMessage conversations with Slack channels. However, the linear time structure of iMessage maps to the alphabetical ordering of Slack channels. Conceptually, mapping to Slack's time structure would make more sense. This may be because the algorithm over emphasizes the conversation-view/channels-view, in which the linear time of iMessage does match with the alphabetical structure of Slack channels. However, we suspect that this is a relatively small change in cost. Lower scoring pairs include code-editor and MacOS Finder,

[6]Analogy Viewer: https://belidor-specification.github.io/

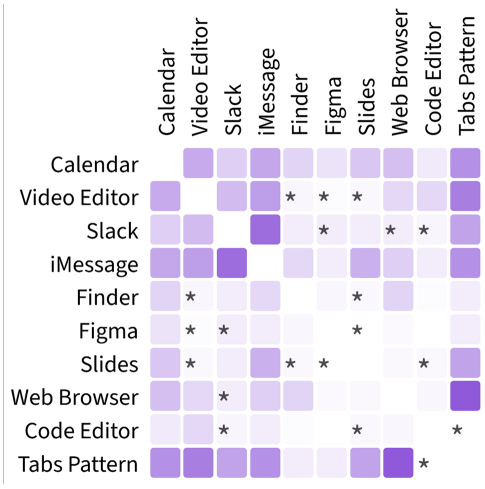


Figure 12: Results from applying the analogy algorithm to every pairing of 10 specifications. Darker cells indicate a stronger conceptual analogy. Asterisks denote pairings that timed out with fewer than four iterations, suggesting they did not adequately navigate the search space in time.

web browser and slides, and MacOS Finder and Slack. Intuitively, these pairs are different from one another, and accordingly the algorithm does not surface a good match. The results of the pairing specifications with tabs pattern are discussed in Section 5.4.

In the following sections, we use case studies to demonstrate how Belidor's semantics and syntax can help highlight and explain user interface analogies.

5.1.3 Algorithm Limitations. The presented algorithm is based on an off-the-shelf graph edit distance that eventually produces an optimal edit path. However, due to the computational complexity of the GED problem, we often cut the algorithm's exploration short, resulting in an approximate answer. One particular issue is that the NetworkX algorithm uses depth-first search, meaning that it can get trapped in suboptimal branches of the search trees for a long time, possibly before we terminate the search. GED algorithms are an active area of research with more sophisticated algorithms to produce approximate results more quickly [15].

5.2 Calendar and Video Editor: A Common Structural Backbone

Belidor foregrounds user interface structure because interactive systems often build on a few conceptual structures. Therefore, if two UIs use the same kind of structure in similar ways, we should expect them to be in strong analogy with one another, even if they are meant for different tasks.

5.2.1 Timelines as the backbone of the analogy. Video editors like DaVinci Resolve, Adobe Premiere and Final Cut Pro are sophisticated creativity support tools with a high skill ceiling. They feature various tools and views to support complex workflows. In contrast, most knowledge workers use a digital calendar every day. While calendars can be complex, in most cases they simply help people synchronize their schedules by coordinating events.

These two applications are used by different people in different contexts. However, they share important structural similarities: both involve presenting and moving blocks on a timeline. Or, in Belidor's terms, both have a linear structure (`time`, `editor-timeline`) with covers (`events`, `videos`) that can be directly manipulated (`move-event`, `move-video`) in presentation layer (`week-view`, `editor-view`) (Figure 13). Aligning the two specifications makes the analogy between the user interfaces obvious (Figure 13). There are nonetheless a few subtle differences. In the presentation layer, the temporal structure is mapped to a *vertical* stack in the calendar but a *horizontal* stack in the video editor. This change is visually significant but structurally inconsequential. Meanwhile, although both `days` and `tracks` are mapped to a cluster encodings, they conceptually differ because `days` cover the timeline whereas `tracks` are just groups. This means that the `days` and `tracks` are mostly similar in how they are used for presentation, but have different meanings. One more significant structural difference is that `videos` have a lot more internal structure than `events`: the former has its own internal timeline affecting images, while the latter only has a few properties like the event's name and its participants. However, this difference does not impact the analogy, since it does not affect either concept's behavior as a cover.

Each of these Belidor specifications could be expanded to include more features, such as the calendar's day and month views, or the video editor's media browser and video player. Because Belidor supports incremental specification (Section 3.8.2), we can focus on these two sub-specifications without much grief.

5.2.2 Designing features based on the analogy. The proposed analogy highlights the conceptual similarity between the two user interfaces, but in SMT's terms, they also have many "alignable differences". This lets us transfer ideas from one UI to the other.

For example, Google Calendar has a notion of appointment slots, which serve as a placeholder for other users to place an event on the calendar. In a video editor, the user could add labeled placeholder clips in the timeline during their initial rough cut. For example, they might leave a placeholder for B-roll to visually pace an interview in an initial rough cut. Later, when triaging B-roll footage, the user can bring up a window listing every B-roll placeholder they used in the rough cut (Figure 15).

Meanwhile, DaVinci Resolve features "adjustment clips", which act like a clip in the timeline and edit the parameters of every clip in the same temporal region. This is often used to apply a consistent color grade across multiple clips. This feature could be ported to a calendar application to set meeting parameters (such as location or video-conferencing links) for multiple meetings over a range of time. This might be helpful if the user wishes to set all of the meetings in the morning to use the same video conferencing link, or for the user to decline meetings during their time off (Figure 14).

5.2.3 Results from the Analogy Algorithm. The algorithm produces a strong match between these two interfaces. As described in this section, it matches the calendar's time structure with the editor's timeline. Notably, the video editor specification was written to let the user open multiple editors using the `group foreach>` syntax, unlike the calendar. Despite this difference, the algorithm still

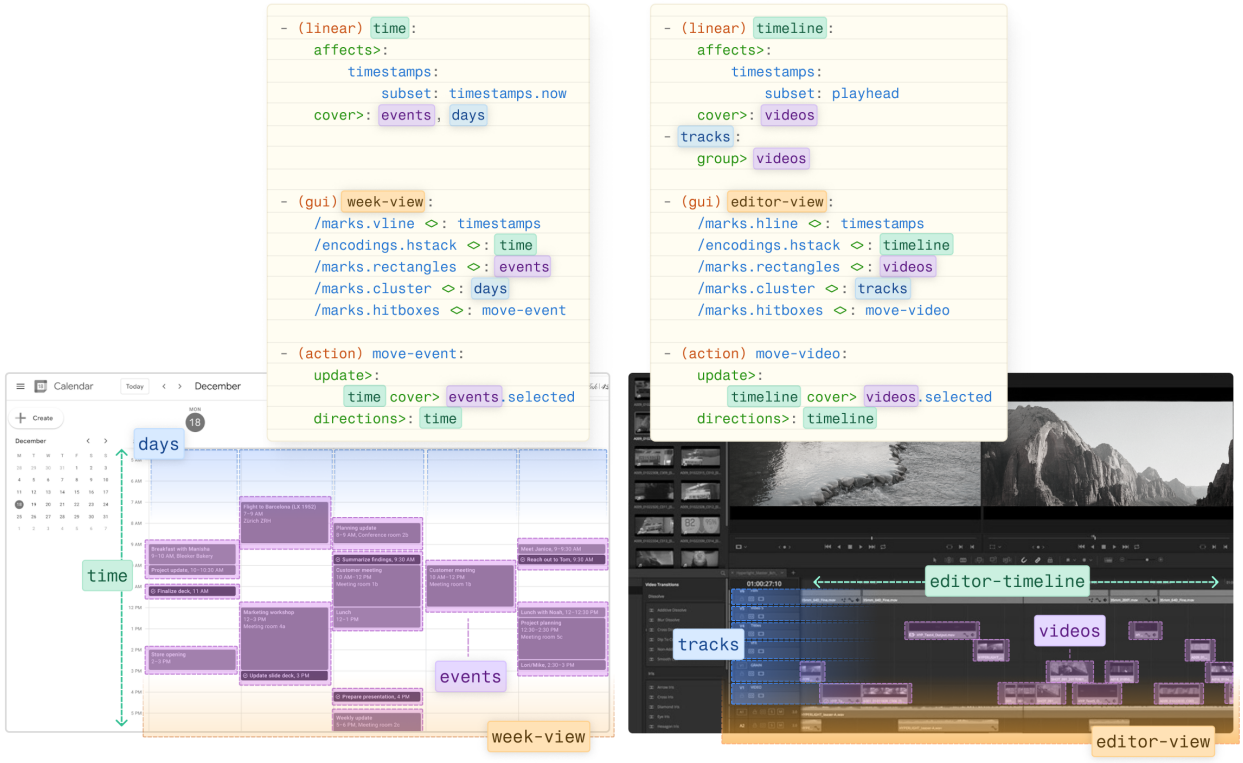


Figure 13: Belidor analogy for a calendar (Google Calendar) and a video editor (DaVinci Resolve). Each have views that present a linear structure (time/timeline) and clustering (days/tracks). Both have covers for the temporal structure (events/videos). Both can directly manipulate events/videos along the time/timeline.

matched the analogy. This illustrates how Belidor's incrementally specification property allows differently written specifications to be treated similarly (Section 3.8.2). In this case, adding a structural change like making part of the application a component does not necessarily interfere with the algorithm's results.

5.3 Figma and Powerpoint: Multiple Overlaps

Interface designers use Figma to make UI mockups, and knowledge workers use PowerPoint to make slides. They are analogous in at least one obvious way: both let users directly manipulate objects on a two dimensional canvas. This analogy is apparent in presentation and in behavior. Of course, Belidor can describe this analogy, but when we wrote the specification for Figma, we realized that its prototyping feature is analogous to PowerPoint's presentation feature (Figure 16). Both let users construct a set of frames/slides connected with a structure (prototype links/slide sequence), and navigate a primary view along the structure (prototype view/presentation slide). Evidently, users have already noticed this analogy: people have used Figma as a presentation tool for many years, and Figma has recently developed their own presentation product [35]. Unsurprisingly, the interaction for specifying links is the same as in their main product.

Notably, this presentation analogy is largely independent from the direct manipulation analogy. This shows how Belidor can surface multiple independent analogies between systems.

5.3.1 Results from the Analogy Algorithm. Terminating the algorithm after two hours resulted in no meaningful analogy. Importantly, it did not reach the iteration threshold, meaning the algorithm may have gotten stuck in a sub-optimal branch. To validate this hypothesis, we can compare the edit cost of the computed analogy with the cost of our preferred analogy (matching canvases, and matching presentation with Figma prototyping). When seeded with our expected matches, we compute an edit cost that is 9.5% lower than the computed analogy from Figure 12. Our preferred analogy also scores a 37% conceptual match, whereas the computed analogy has a 0% conceptual match. Therefore, the initial computation was prematurely terminated and carrying the algorithm to completion may have resulted in our more optimal analogy.

5.4 Web Browser and CAD: Emerging Design Patterns

Different tools might be conceptually different from one another, but use similar strategies as part of their interface design. Then, they are analogous in that they use a common design pattern. For example, web browsers are different from 3D Computer-Assisted Design (CAD) applications. Navigating between web pages is structurally

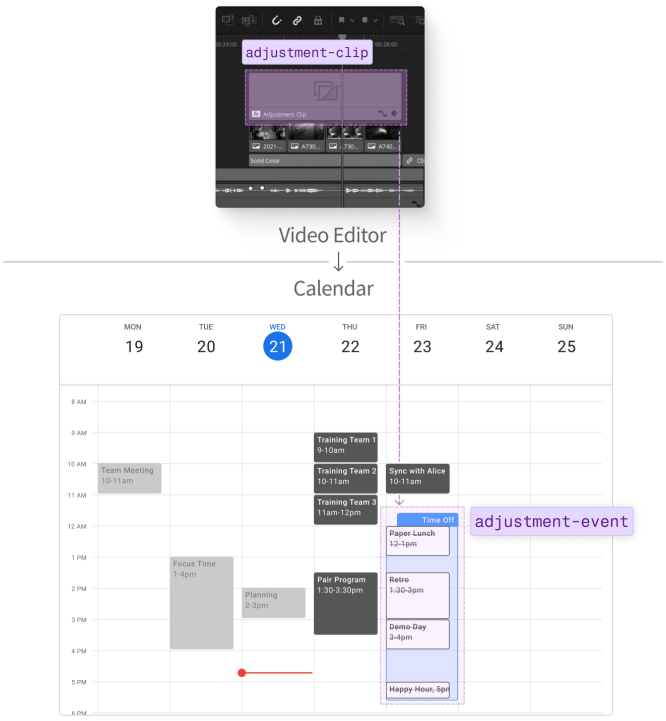


Figure 14: Video editor adjustment layer analogy. The user can create an adjustment event to change the settings for all events that overlap it, for example to dismiss them. The adjustment behaves like an event, so its start and end can be adjusted.

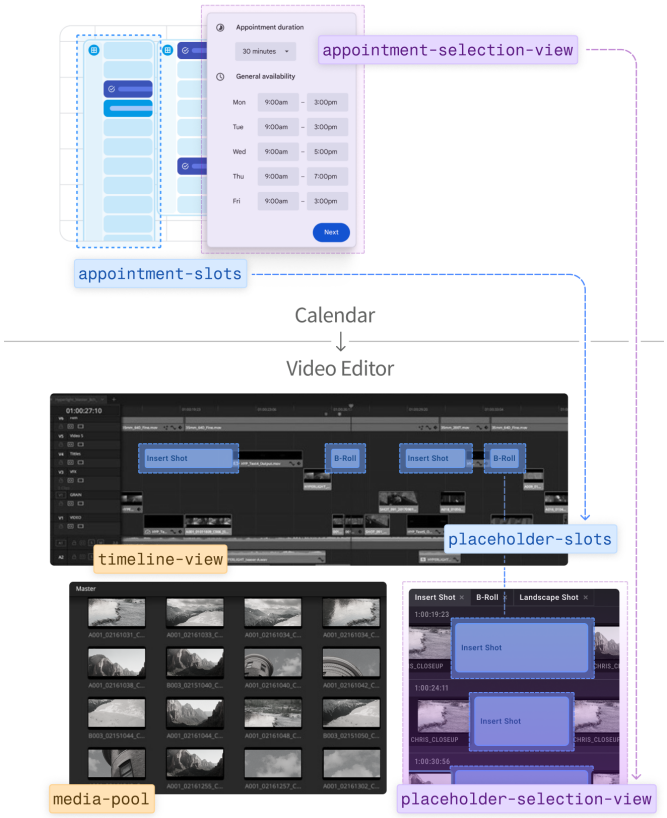


Figure 15: Calendar appointment analogy. The user adds placeholder clips to the timeline. Later, when navigating the media pool, they find an insert shot and open a window listing all placeholder clips in the timeline's context. This helps them choose where to add the shot without leaving the media pool.

different to building and assembling 3D objects from sequences of operations. As a result the core specifications are also different from one another. However, both applications use tabs to let the user jump between different views —presumably to solve the same screen real-estate problem. This is summarized in specifications from Figure 17. The content (`webpages/files`) defines its own view; a tabs-view presents both the content (`webpages.open/files`).

Once again, Belidor's incremental specification lets authors focus on the tabs functionality. Since the CAD's 3D structure is not relevant, it is not specified. Placing the specifications side-by-side emphasizes the similarity between their tabs. In fact, by generalizing the terms used in both specifications, a specification for the tabs design pattern emerges from the comparison. This means that Belidor can describe design patterns as standalone Belidor specifications. Then, using the same analogy mechanisms as before, we can find specifications that use the same design pattern. Cognitive Science has long studied how analogies can lead to abstractions, in this case in the form of a design pattern [25, 62]. The pattern specification can use the generic terms designers use when discussing tabs in the abstract.

```
# Tabs design pattern on a generic subject
- subjects:
    group foreach>:
        - (gui) /view

- (gui) tabs-view:
    /marks <>:
      - subjects
      - (action) change-tab:
          update>: subjects.active subset>
              subjects
    /encoding.stack <>:
      (linear) tabs-order:
        affects>: subjects
- (gui) main-view:
    /marks <>: subjects.active/view
```

By searching for applications using this pattern, we can find UIs that are not obviously to be using tabs. For example, presentation tools like PowerPoint have a primary editing view and a view presenting all of the other slides. This design is not typically described

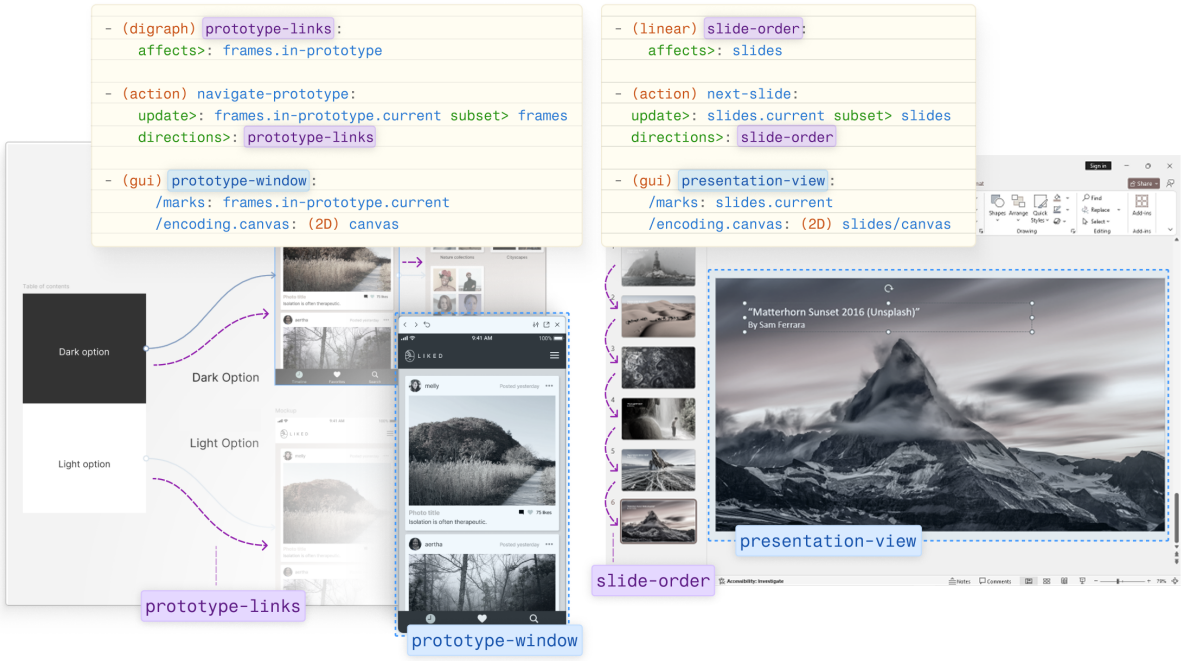


Figure 16: In addition to their canvas editing, Figma and PowerPoint overlap in the prototyping/presentation functionality. Each have a view (`prototype-window`presentation-view`) showing a specific item (current prototype frame/current slide), that the user can navigate by following a structure (`prototype-links`slides-order`)

as tabs, but it is structurally and functionally the same. Slightly less conventionally, the iPad application dock also functions like tabs: each app has its own view, and the dock presents a linear stack of shortcuts to open the view.

This more expansive view of tabs helps transfers of ideas. For example, the iPad can show multiple applications on the same window. The same interaction technique could be used in any of the other applications to show more than one view at a time.

5.4.1 Results from the Analogy Algorithm. The specification for the tabs pattern matched with a number of the other applications. It successfully matched with web browser tabs and matched with tab-like features such as channels in Slack, conversations in iOS Messages, and it the slides in PowerPoint. Meanwhile, MacOS Finder's tabs were not denoted in our specification and therefore were not found. The code-editor does have tabs, but the algorithm seemed stuck in an unfruitful branch before terminating.

Unconventional matches were also found. For example, in the calendar and video editor, although the tabs object was not matched, the actions and structure around tabs were matched to the timeline: moving along the timeline (eg. time passing in the calendar) is matched with changing tabs. This is structurally a reasonable match, and might be generative in its own way. For example, the appointment concept in the calendar might be applied to tabs, where placeholder tabs could be used as slots to be populated. This could be used with Large Language Model based search: instead of haphazardly opening tabs from a prompt, the user could denote what content they want in specific tabs before completing the search.

6 Discussion and Future Work

We presented Belidor, a declarative specification language for interactive systems. It reifies the structures underlying user interfaces as a first-class language construct to describe the conceptual model, presentation, and behavior of interactive systems. As a result, conceptual structures that help present information (ie. with encodings), or facilitate interaction (eg. navigation) are made explicit. We demonstrated that Belidor serves as an effective representation for making analogies both manually with case studies and computationally with a simple algorithm. The analogies surfaced by Belidor help designers use ideas across user interfaces, whether they are obviously similar, or seemingly unrelated. In the latter case, Belidor can explicitly denote the overlap between the two UIs, for example to describe common design patterns.

This paper has demonstrated that Belidor is a parsimonious and expressive representation of interactive systems which can help find and describe analogies. We now discuss future work for Belidor and its possible applications.

6.1 Designing with Belidor Analogies

Although Belidor can help surface distant analogies, this paper does not show that they are made *accessible* to the average designer. Evaluating UI toolkits, let alone theoretical frameworks, is notoriously difficult [53]. Olsen argues that language contributions break the assumptions of a user study: for example, people can not be expected to "walk up and use" a language like they might pick up a well designed GUI. Instead, we expect designers would need time to internalize the language's syntax and semantics. Therefore,

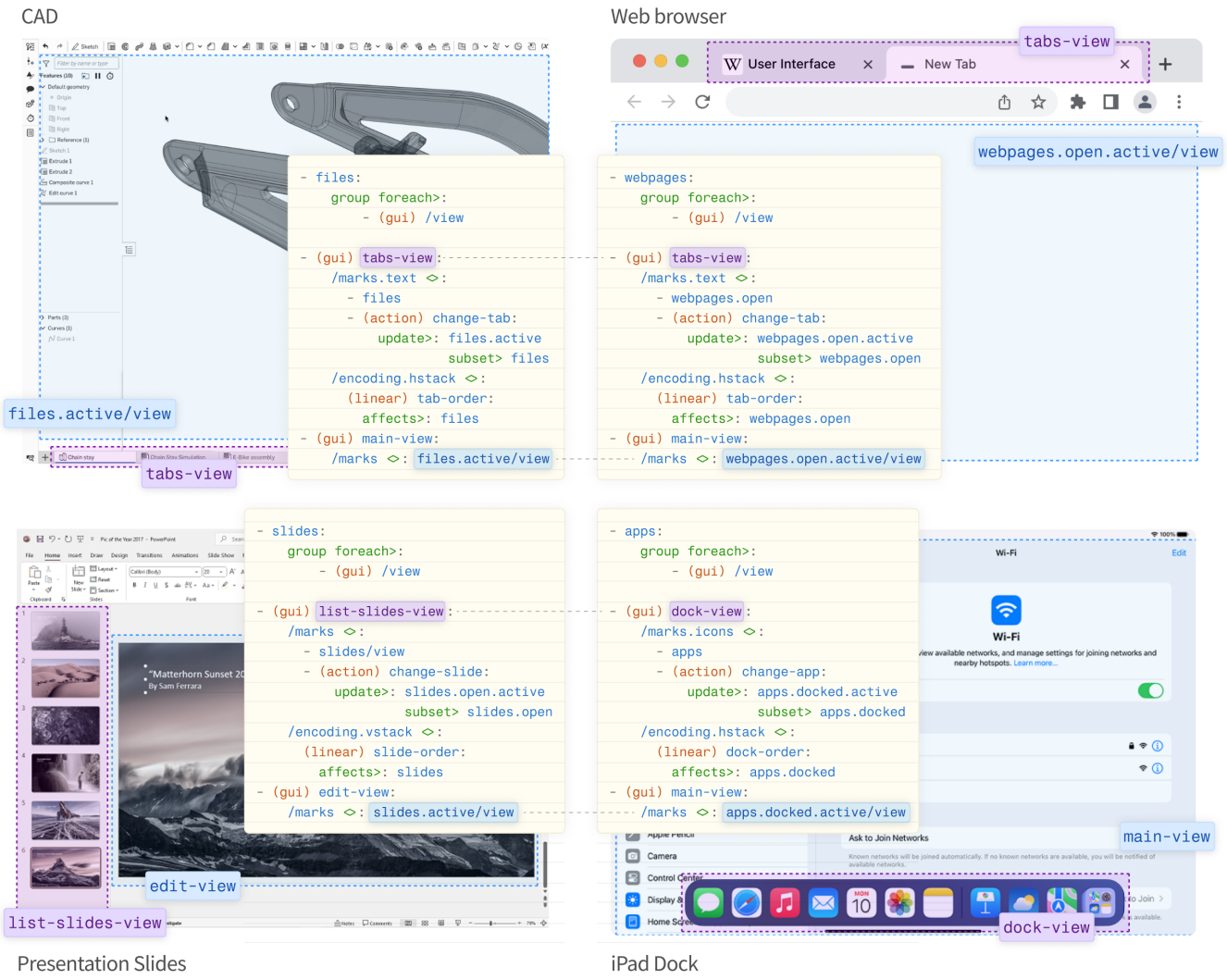


Figure 17: Four applications that use a tab pattern. Top-left: CAD application presents files as tabs (Onshape). Top-right: A web browser uses tabs to switch between webpages (Chrome). Bottom-left: the list of slides in a presentation application are structurally equivalent to tabs (PowerPoint). Bottom-right: The iPad dock functions like tabs for applications.

studying how people use a language requires longer term upfront training and results in rich qualitative observations, both of which are beyond the scope of this paper.

In future work, we intend to study how designers use Belidor analogies in an iterative design process. Given a design challenge, participants can be tasked with writing the Belidor specification of an initial design to query a database of UIs for analogies. We can then study how participants write Belidor specifications and how they use the resulting analogies in their design process. In particular, we want to investigate how writing in Belidor and reasoning through analogies helps designers reflect on their design [58].

It may seem unlikely that interface designers, accustomed to working with visual sketches and mock-ups, might choose to write textual specifications in their design process. However, the same could have been said about visualization before the Grammar of Graphics, when GUIs like Excel were more popular for the task. And yet, scientists and visualization experts today prefer declarative languages like ggplot to GUIs because text notations are more powerful and expressive. By studying how people write Belidor, we can better understand what its expressivity and parsimony can uniquely afford designers.

6.2 Building UIs with Belidor Specifications

Belidor is not intended as a UI framework for implementing interactive systems, but it is not far off — its entity-relationship models can be implemented as a database, and Belidor queries can be interpreted as relational queries. The presentation layer is inspired by Bluefish, so its layout algorithm could potentially be adapted to render the presentation layer [56].

However, behavior in Belidor is harder to translate into an implementation. As discussed in Section 3.8.3, we decided against describing input bindings, state machines and computation because the resulting increased complexity did not comparatively improve analogies. However, the gap is not insurmountable. Card et al.'s analysis demonstrates that input devices have their own structure which could map to the Belidor's structure of actions [12]. The current syntax can already model rudimentary state machines: a directed graph of transitions affects a set of states, and the current state is a subset that navigates the transition structure. For example, this pattern is used to describe how the user navigates between different views. To describe modes—the condition under which an action can take place—Belidor could use the `when` keyword to indicate the condition required for an action to occur. For example, keyboard shortcut actions could be guarded by which view is active.

A UI framework built on Belidor's semantics could help developers rapidly iterate on structure-level changes. For example, Min and Xia [49] demonstrate this with a framework for overview-detail applications, in which a selected item is shown in detail (eg. a restaurant) while other items are shown in an overview (eg. a restaurant list). The framework's semantics help developers trivially change the overview representation from a list to a table or a map. Similarly, a framework using Belidor's semantics could facilitate changing how a structure is used across the conceptual presentation behavior layers. This would help evaluate a broader range of possible designs with minimal cost.

6.3 Design Moves Beyond Analogies

Using an analogy is a design move; it "moves" a design from one point in the design space of possible user interfaces to another. We believe that Belidor's semantics have the potential to afford a broader range of design moves. For example, Belidor can help designers iterate on the structures underlying their user interface. In the messaging example, iOS Message designers added a group structure to pin certain conversations. This is a structural design move — designers found a user problem that could be solved by adding structure to conversations, and then designed how it translated to presentation. By providing a notation for interface structure, Belidor can help designers externalize and therefore reflect on their structural design moves.

In HCI, many system design contributions can be understood as structural design moves. For example, Toolglasses and Magic Lenses contributed interaction techniques based on covers of the GUI (toolglasses) [7]. In another context, research aimed at developing screen reader applications finds ways to represent more concepts as audible marks, and develops structures to help present and navigate them. For example, Olli [9] contributes a tree structure to the navigation of data charts, which Data Navigator [20] builds on by generalizing the tree as an arbitrary directed graph. Finally, Interaction Substrates emphasize the importance of structure when designing creativity support tools for power and simplicity [45].

Ultimately, our goal is to help designers discuss and reflect on the interactive systems they design. This may seem at odds with the formal approach taken in this paper: after all, formal specifications are simply too cumbersome for informal conversations. However, designing a formal language forces its semantics to be robust and consistent, providing a stable foundation that continues to pay off when the syntax is relaxed. This is the story of Unified Modeling Language (UML): although it has a strict formal specification, software engineers loosen and appropriate the notation at the whiteboard to collaboratively express and debate possible architectures [17, 47]. Although professionals do not use the formalism directly, it nonetheless provided a foundation they could appropriate for their practice.

In summary, we contend that interface designers already engage in structural design, but lack a suitable language to express these ideas. We envision that Belidor, or a future derivative, can serve as a representation that fosters structural design moves and helps designers better explore the design space of user interfaces.

Acknowledgments

We thank the reviewers for their thoughtful feedback, Brian Hempel for guidance on programming language design and help with the analogy algorithm, and Camille Gobert and Josh Pollock for helpful discussions on Belidor's language design. This work was partially supported by Grant No. NSF IIS-1845900.

References

[1] Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel, and Patrick Martineau. 2015. An Exact Graph Edit Distance Algorithm for Solving Pattern Recognition Problems. In *Proceedings of the International Conference on Pattern Recognition Applications and Methods - Volume 1* (Lisbon, Portugal) *(ICPRAM 2015)*. SCITEPRESS - Science and Technology Publications, Lda, Setubal, PRT, 271–278. doi:10.5220/0005209202710278

[2] Anand Agarawala and Ravin Balakrishnan. 2006. Keepin' it real: pushing the desktop metaphor with physics, piles and the pen. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Montréal, Québec, Canada) *(CHI '06)*. Association for Computing Machinery, New York, NY, USA, 1283–1292. doi:10.1145/1124772.1124965

[3] Matthew T Beaudouin-Lafon, Jane L E, and Haijun Xia. 2023. Color Field: Developing Professional Vision by Visualizing the Effects of Color Filters. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23)*. Association for Computing Machinery, New York, NY, USA, Article 101, 16 pages. doi:10.1145/3586183.3606828

[4] Farnaz Behrang, Steven P. Reiss, and Alessandro Orso. 2018. GUIfetch: supporting app design and development through GUI search. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems* (Gothenburg, Sweden) *(MOBILESoft '18)*. Association for Computing Machinery, New York, NY, USA, 236–246. doi:10.1145/3197231.3197244

[5] Edward O Benson and David R Karger. 2013. Cascading tree sheets and recombinant HTML: better encapsulation and retargeting of web content. In *Proceedings of the 22nd international conference on World Wide Web*. Association for Computing Machinery, New York, NY, United States, 107–118.

[6] Gilbert Louis Bernstein and Scott Klemmer. 2014. Towards responsive retargeting of existing websites. In *Adjunct Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology* (Honolulu, Hawaii, USA) *(UIST '14 Adjunct)*. Association for Computing Machinery, New York, NY, USA, 119–120. doi:10.1145/2658779.2658805

[7] Eric A. Bier, Maureen C. Stone, Ken Pier, William Buxton, and Tony D. DeRose. 1993. Toolglass and magic lenses: the see-through interface. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques* (Anaheim, CA) *(SIGGRAPH '93)*. Association for Computing Machinery, New York, NY, USA, 73–80. doi:10.1145/166117.166126

[8] Alan F. Blackwell. 2006. The reification of metaphor as a design tool. *ACM Trans. Comput.-Hum. Interact.* 13, 4 (Dec. 2006), 490–530. doi:10.1145/1188816.1188820

[9] Matt Blanco, Jonathan Zong, and Arvind Satyanarayan. 2022. Olli: An Extensible Visualization Library for Screen Reader Accessibility. https://vis.csail.mit.edu/pubs/olli

[10] Grady Booch, James Rumbaugh, and Ivar Jacobson. 2005. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, Boston, MA, United States.

[11] Sara Bunian, Kai Li, Chaima Jemmali, Casper Harteveld, Yun Fu, and Magy Seif Seif El-Nasr. 2021. VINS: Visual Search for Mobile User Interface Design. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*

(Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 423, 14 pages. doi:10.1145/3411764.3445762

[12] Stuart K. Card, Jock D. Mackinlay, and George G. Robertson. 1991. A morphological analysis of the design space of input devices. *ACM Trans. Inf. Syst.* 9, 2 (April 1991), 99–122. doi:10.1145/123078.128726

[13] Hernan Casakin and Gabriela Goldschmidt. 1999. Expertise and the use of visual analogy: implications for design education. *Design Studies* 20, 2 (1999), 153–175. doi:10.1016/S0142-694X(98)00032-5

[14] Yam San Chee. 1993. Applying Gentner's theory of analogy to the teaching of computer programming. *International Journal of Man-Machine Studies* 38, 3 (1993), 347–368. doi:10.1006/imms.1993.1016

[15] Qihao Cheng, Da Yan, Tianhao Wu, Zhongyi Huang, and Qin Zhang. 2025. Computing Approximate Graph Edit Distance via Optimal Transport. *Proc. ACM Manag. Data* 3, 1, Article 23 (Feb. 2025), 26 pages. doi:10.1145/3709673

[16] Stéphane Conversy. 2011. Improving usability of interactive graphics specification and implementation with picking views and inverse transformation. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, IEEE Computer Society, Washington, DC, United States, 153–160.

[17] Uri Dekel and James D. Herbsleb. 2007. Notation and representation in collaborative object-oriented design: an observational study. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (Montreal, Quebec, Canada) *(OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 261–280. doi:10.1145/1297027.1297047

[18] David J. Duke and Michael D. Harrison. 1993. Abstract interaction objects. In *Computer Graphics Forum*, Vol. 12. Wiley Online Library, Wiley and Eurographics, New York, NY, USA, 25–36. Issue 3.

[19] Kevin Dunbar. 1997. How scientists think: On-line creativity and conceptual change in science. In *Creative thought: An investigation of conceptual structures and processes*, J. Vaid T. B. Ward, S. M. Smith (Ed.). American Psychological Association, 461–493. doi:10.1037/10227-017

[20] Frank Elavsky, Lucas Nadolskis, and Dominik Moritz. 2023. Data navigator: an accessibility-centered data navigation toolkit. *IEEE transactions on visualization and computer graphics* 30, 1 (2023), 803–813.

[21] International Organization for Standardization. 1995. Prolog. https://www.iso.org/standard/21413.html

[22] Dedre Gentner. 1983. Structure-mapping: A theoretical framework for analogy. *Cognitive science* 7, 2 (1983), 155–170.

[23] Dedre Gentner. 1989. The mechanisms of analogical learning. *Similarity and analogical reasoning* 199 (1989), 199–241.

[24] Dedre Gentner, Brian Bowdle, Phillip Wolff, Consuelo Boronat, et al. 2001. Metaphor is like analogy. In *The analogical mind: Perspectives from cognitive science*. MIT Press, 199–253.

[25] Dedre Gentner and Christian Hoyos. 2017. Analogy and abstraction. *Topics in cognitive science* 9, 3 (2017), 672–693.

[26] Dedre Gentner and Michael Jeziorski. 1993. *The shift from metaphor to analogy in Western science*. Cambridge University Press, 447–480.

[27] Dedre Gentner, Jeffrey Loewenstein, and Leigh Thompson. 2004. Analogical encoding: Facilitating knowledge transfer and integration. In *Proceedings of the annual meeting of the Cognitive Science Society*, Vol. 26. Cognitive Science Society, 586—-597. Issue 26.

[28] Mary L. Gick and Keith J. Holyoak. 1980. Analogical problem solving. *Cognitive Psychology* 12 (1980), 306–355. Issue 3. doi:10.1016/0010-0285(80)90013-4

[29] Charles Goodwin. 1994. Professional Vision. *American Anthropologist* 96, 3 (1994), 606–633. doi:10.1525/aa.1994.96.3.02a00100

[30] Thomas R. G. Green and Marian Petre. 1996. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing* 7, 2 (1996), 131–174.

[31] Aric Hagberg, Pieter J Swart, and Daniel A Schult. 2008. *Exploring network structure, dynamics, and function using NetworkX*. Technical Report. Los Alamos National Laboratory (LANL), Los Alamos, NM (United States).

[32] Frank Halasz and Thomas P. Moran. 1982. Analogy considered harmful. In *Proceedings of the 1982 Conference on Human Factors in Computing Systems* (Gaithersburg, Maryland, USA) *(CHI '82)*. Association for Computing Machinery, New York, NY, USA, 383–386. doi:10.1145/800049.801816

[33] Forrest Huang, John F. Canny, and Jeffrey Nichols. 2019. Swire: Sketch-based User Interface Retrieval. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/3290605.3300334

[34] Edwin Hutchins. 1987. *Metaphors for interface design*. ICS Report.

[35] Figma Inc. 2024-2025. Figma Slides. https://www.figma.com/slides/

[36] Meta Inc. 2013. React. https://react.dev/

[37] Daniel Jackson. 2021. *The essence of software: Why concepts matter for great design*. Princeton University Press.

[38] Yue Jiang, Changkong Zhou, Vikas Garg, and Antti Oulasvirta. 2024. Graph4GUI: Graph Neural Networks for Representing Graphical User Interfaces. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 988, 18 pages. doi:10.1145/3613904.3642822

[39] David J Kasik. 1982. A user interface management system. In *Proceedings of the 9th annual conference on Computer graphics and interactive techniques*. Association for Computing Machinery, 99–106.

[40] Glenn E. Krasner and Stephen T. Pope. 1988. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *J. Object Oriented Program.* 1, 3 (Aug. 1988), 26–49.

[41] Ranjitha Kumar, Jerry O. Talton, Salman Ahmad, and Scott R. Klemmer. 2011. Bricolage: example-based retargeting for web design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vancouver, BC, Canada) *(CHI '11)*. Association for Computing Machinery, New York, NY, USA, 2197–2206. doi:10.1145/1978942.1979262

[42] Luis A. Leiva, Asutosh Hota, and Antti Oulasvirta. 2021. Enrico: A Dataset for Topic Modeling of Mobile UI Designs. In *22nd International Conference on Human-Computer Interaction with Mobile Devices and Services* (Oldenburg, Germany) *(MobileHCI '20)*. Association for Computing Machinery, New York, NY, USA, Article 9, 4 pages. doi:10.1145/3406324.3410710

[43] Jeffrey Loewenstein. 2010. Chapter 4 - How One's Hook Is Baited Matters for Catching an Analogy. In *The Psychology of Learning and Motivation: Advances in Research and Theory*, Brian H. Ross (Ed.). Psychology of Learning and Motivation, Vol. 53. Academic Press, 149–182. doi:10.1016/S0079-7421(10)53004-4

[44] Yuwen Lu, Alan Leung, Amanda Swearngin, Jeffrey Nichols, and Titus Barik. 2025. Misty: UI Prototyping Through Interactive Conceptual Blending. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. Association for Computing Machinery, New York, NY, USA, Article 1108, 17 pages. doi:10.1145/3706598.3713924

[45] Wendy E. Mackay and Michel Beaudouin-Lafon. 2025. Interaction Substrates: Combining Power and Simplicity in Interactive Systems. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. Association for Computing Machinery, New York, NY, USA, Article 687, 16 pages. doi:10.1145/3706598.3714006

[46] Richard Mander, Gitta Salomon, and Yin Yin Wong. 1992. A "pile" metaphor for supporting casual organization of information. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Monterey, California, USA) *(CHI '92)*. Association for Computing Machinery, New York, NY, USA, 627–634. doi:10.1145/142750.143055

[47] Nicolas Mangano, Thomas D LaToza, Marian Petre, and André Van der Hoek. 2014. How software designers interact with sketches at the whiteboard. *IEEE Transactions on Software Engineering* 41, 2 (2014), 135–156.

[48] Damien Masson, Young-Ho Kim, and Fanny Chevalier. 2025. Textoshop: Interactions Inspired by Drawing Software to Facilitate Text Editing. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. Association for Computing Machinery, New York, NY, USA, Article 1087, 14 pages. doi:10.1145/3706598.3713862

[49] Bryan Min and Haijun Xia. 2025. Meridian: A Design Framework for Malleable Overview-Detail Interfaces. In *Proceedings of the 38th Annual ACM Symposium on User Interface Software and Technology (UIST '25)*. Association for Computing Machinery, New York, NY, USA, Article 200, 14 pages. doi:10.1145/3746059.3747654

[50] Mohammad Amin Mozaffari, Xinyuan Zhang, Jinghui Cheng, and Jin L.C. Guo. 2022. GANSpiration: Balancing Targeted and Serendipitous Inspiration in User Interface Design with Style-Based Generative Adversarial Network. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 537, 15 pages. doi:10.1145/3491102.3517511

[51] Jeffrey Nichols, Brad A. Myers, and Brandon Rothrock. 2006. UNIFORM: automatically generating consistent remote control user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Montréal, Québec, Canada) *(CHI '06)*. Association for Computing Machinery, New York, NY, USA, 611–620. doi:10.1145/1124772.1124865

[52] Laura R Novick and Keith J Holyoak. 1991. Mathematical problem solving by analogy. *Journal of experimental psychology: Learning, memory, and cognition* 17, 3 (1991), 398.

[53] Dan R. Olsen. 2007. Evaluating user interface systems research. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology* (Newport, Rhode Island, USA) *(UIST '07)*. Association for Computing Machinery, New York, NY, USA, 251–258. doi:10.1145/1294211.1294256

[54] Fabio Paterno. 1999. *Model-based design and evaluation of interactive applications*. Springer, London, England.

[55] Paulo Pinheiro da Silva. 2000. User interface declarative models and development environments: A survey. In *International Workshop on Design, Specification, and Verification of Interactive Systems*. Springer, 207–226.

[56] Josh Pollock, Catherine Mei, Grace Huang, Elliot Evans, Daniel Jackson, and Arvind Satyanarayan. 2024. Bluefish: Composing Diagrams with Declarative Relations. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology* (Pittsburgh, PA, USA) *(UIST '24)*. Association for Computing Machinery, New York, NY, USA, Article 23, 21 pages. doi:10.1145/3654777.3676465

[57] Josh Pollock and Arvind Satyanarayan. 2025. GoFish: a Grammar of More Graphics! *IEEE Transactions on Visualization and Computer Graphics* PP (12 2025), 1–11. doi:10.1109/TVCG.2025.3634250

[58] Donald Schön. 1983. *The Reflective Practitioner: How Professionals Think in Action*. Basic Books.

[59] 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* (Marina del Rey, California, USA) *(UIST '94)*. Association for Computing Machinery, New York, NY, USA, 133–135. doi:10.1145/192426.192479

[60] Amanda Swearngin, Mira Dontcheva, Wilmot Li, Joel Brandt, Morgan Dixon, and Amy J. Ko. 2018. Rewire: Interface Design Assistance from Examples. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3173574.3174078

[61] Tableau 2003-2026. Tableau: A Visualization Grammer. https://www.tableau.com/

[62] Mark Tunner and Gilles Fauconnier. 1995. Conceptual Integration and Formal Expression. *Metaphor and Symbolic Activity* 10, 3 (1995), 183–204.

[63] Kurt VanLehn. 1998. Analogy events: How examples are used during problem solving. *Cognitive Science* 22, 3 (1998), 347–388.

[64] Hadley Wickham. 2016. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, New York, NY, United States. https://ggplot2.tidyverse.org

[65] Leland Wilkinson. 2011. The grammar of graphics. In *Handbook of computational statistics: Concepts and methods*. Springer, 375–414.

[66] Jason Wu, Amanda Swearngin, Xiaoyi Zhang, Jeffrey Nichols, and Jeffrey P. Bigham. 2023. Screen Correspondence: Mapping Interchangeable Elements between UIs. arXiv:2301.08372 [cs.HC] https://arxiv.org/abs/2301.08372

[67] Jason Wu, Xiaoyi Zhang, Jeff Nichols, and Jeffrey P Bigham. 2021. Screen Parsing: Towards Reverse Engineering of UI Models from Screenshots. In *The 34th Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '21)*. Association for Computing Machinery, New York, NY, USA, 470–483. doi:10.1145/3472749.3474763

[68] Zhiping Zeng, Anthony K. H. Tung, Jianyong Wang, Jianhua Feng, and Lizhu Zhou. 2009. Comparing stars: on approximating graph edit distance. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 25–36. doi:10.14778/1687627.1687631