Varv: Reprogrammable Interactive Software as a Declarative Data Structure

Marcel Borowski
marcel.borowski@cs.au.dk
Aarhus University
Aarhus, Denmark

Luke Murray
lsmurray@mit.edu
MIT CSAIL
Cambridge, United States

Rolf Bagge
rolf@cavi.au.dk
Aarhus University
Aarhus, Denmark

Janus Bager Kristensen
jbk@cavi.au.dk
Aarhus University
Aarhus, Denmark

Arvind Satyanarayan
arvindsatya@mit.edu
MIT CSAIL
Cambridge, United States

Clemens N. Klokmose
clemens@cs.au.dk
Aarhus University
Aarhus, Denmark

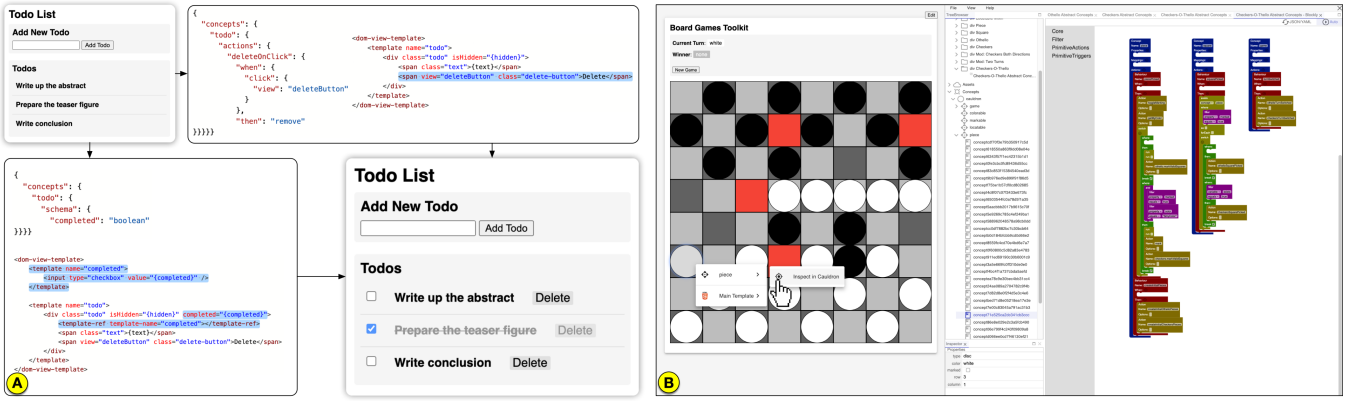


Figure 1: Varv Examples: (a) A todo list web application that is inherently extensible. Here, a basic todo list is extended with the ability to complete and delete todos by adding two new concept definitions and new modified template definitions. (b) A board game toolkit that defines abstractions for board game logic. The games "Checkers" and "Othello" were implemented with the toolkit and then merged into a new "Checkers-O-Thello" game with the addition of a short concept definition. As Varv applications are represented as data structures, higher-level tooling can be developed including a block-based editor (right), an inspector to go from an element in the view to the corresponding template or data (context menu to the left), and a data inspector for live editing application state (middle).

ABSTRACT

Most modern applications are immutable and turn-key despite the acknowledged benefits of empowering users to modify their software. Writing extensible software remains challenging, even for expert programmers. Reprogramming or extending existing software is often laborious or wholly blocked, requiring sophisticated knowledge of application architecture or setting up a development environment. We present Varv, a programming model representing reprogrammable interactive software as a declarative data structure. Varv defines interactive applications as a set of concepts that consist of a schema and actions. Applications in Varv support incremental modification, allowing users to reprogram through addition and selectively suppress, modify, or add behavior. Users can define high-level concepts, creating an abstraction layer and effectively a domain-specific language for their application domain, emphasizing reuse and modification. We demonstrate the reprogramming and collaboration capabilities of Varv in two case studies and illustrate how the event engine allows for extensive tooling support.

CCS CONCEPTS

• **Human-centered computing** → *Web-based interaction*; *Collaborative interaction*; **User interface programming**; • **Software and its engineering** → *Real-time systems software*; *Abstraction, modeling and modularity.*

KEYWORDS

declarative programming, reprogramming, interactive software, liveness, real-time collaboration

CHI '22, April 29-May 5, 2022, New Orleans, LA, USA

ACM ISBN 978-1-4503-9157-3/22/04…$15.00
https://doi.org/10.1145/3491102.3502064

ACM Reference Format:
Marcel Borowski, Luke Murray, Rolf Bagge, Janus Bager Kristensen, Arvind Satyanarayan, and Clemens N. Klokmose. 2022. Varv: Reprogrammable Interactive Software as a Declarative Data Structure. In *CHI Conference on Human Factors in Computing Systems (CHI '22), April 29-May 5, 2022, New Orleans, LA, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3491102.3502064

1 INTRODUCTION

It has long been acknowledged that most programs are written not by professional software developers but rather by end-users [42] who, for instance, regularly build small computational tools such as creating their own interfaces in spreadsheets. However, most application software is constructed, packaged, and shared as hermetically-sealed turn-key products [34, 56]. As a result, end-users — including professionally trained programmers — have little power to change the applications they use. In their foundational 1977 piece on *Personal Dynamic Media*, Kay and Goldberg envisioned software being *malleable*, so that users could easily redefine and reshape it to suit their idiosyncratic needs [37].

Today, software malleability primarily occurs through scripting (e.g., macOS Automator, iOS Shortcuts, or IFTTT) or add-on extensions (e.g., Firefox, Figma, or Visual Studio Code). While such facilities can yield "customization ecosystems" that increase the value of the application for all users [27], these approaches present a non-trivial burden for both software creators and end-user programmers. Writing *extensible* software is an explicit choice that software creators must make, and requires careful design and architectural decisions that are often untenable for small-scale software creators to consider as the customization ecosystem is not guaranteed to flourish. Moreover, the range of customizations these facilities support is circumscribed by the design of their APIs, thereby presenting a catch-22: it may not be possible to customize particular aspects of an application if the creator did not foresee the possibility of doing so. Finally, the APIs themselves are idiosyncratic and application-specific. As a result, it can be challenging to engage in customizations in a cross-cutting fashion — for instance, porting an extension from one context to another typically amounts to rewriting it from scratch, and extending or composing add-ons together is inconceivable without direct modification of their source code.

In response, we present Varv,[1] a declarative language for reprogrammable interactive software that decouples specification (the *what*) from execution (the *how*). With Varv, users can focus on specifying interactive applications as compositions of *concepts*, or individual units of dynamic functionality. Concepts comprise a *schema*, that specifies the shape and type of the concept's state; *actions*, that describe valid transformations of the state; and, *triggers*, or events that cause transitions between states. Inspired by Vega [66] and Vega-Lite [65], concepts are specified as data structures expressed in JSON (JavaScript Object Notation). The Varv runtime is responsible for all execution concerns, including parsing declarative specifications, assembling the corresponding dataflow graph, and handling event creation and propagation. The runtime also handles bookkeeping associated with storing application state and rendering the resultant interface — Varv is designed to be agnostic to the specific ways these processes occur. As a result, Varv can target a variety of data backends or frontend modalities.

Varv's structured, declarative approach contrasts existing methods for constructing interactive software, which typically involves writing unstructured blobs of imperative code. It yields an *accretive* development process, with applications that are *inherently extensible*. In particular, application developers need no longer write explicit extensibility APIs. Instead, to introduce a new piece of functionality, end-user programmers introduce a new JSON object at runtime. These JSON objects can extend or override existing concept definitions in a straightforward fashion or use a series of *composition operators* to construct new concepts from existing parts. The Varv architecture consolidates new and existing specifications and hot-swaps them to produce a *live programming* experience (i.e., users see changes they make to Varv program specifications reflected immediately). In this way, Varv blurs the boundary between developing the "core" application and extending it, making it possible for users to tinker with interactive functionality incrementally.

We evaluate the feasibility and expressivity of our approach through demonstration [44]. We first instantiate Varv in Webstrates [41], a web-based environment that provides persistence and real-time synchronization of application data (i.e., state). While Webstrates provides our data layer, Codestrates [13] provides a code editing layer on top of web pages that enables instantiating IDE-like tools inside a web app. With this Webstrates-based implementation, we develop two case studies to demonstrate that Varv can be used to author a rich design space of interactive applications. The case studies illustrate the experience of using Varv as a live programming tool for user interfaces, akin to real-time manipulation of HTML and CSS using the browser's built-in developer tools, but now for interactive behavior as well. We show how users can author Varv applications incrementally — one feature at a time, where each feature is implemented as an extension to the application rather than a modification of existing source code — and how concepts can be used to prototype domain-specific languages for developing and composing classes of applications. This implementation required no modifications to Webstrates itself. The case studies also demonstrate the synergies between the two paradigms: using Varv with Webstrates yields a live and collaborative reprogramming experience. However, to illustrate that Varv is agnostic to data storage, we develop two additional prototype implementations, one packaged in Electron [60], an environment that enables local development of Varv applications, and one deployed on Observable [57], a web-based notebook environment for JavaScript.

We, moreover, demonstrate the implications of our approach through a series of prototype sketches of higher-level tooling to support Varv application development. Although Varv applications are specified as JSON-based data structures, we show how this representation facilitates a range of authoring experiences, including visual block-based editing and alternative specification formats such as YAML. Similarly, we show how Varv's declarative representation enables visual inspectors for debugging. With Varv, we set a foundation for malleable software to enable users — who are, for now, proficient in programming — to modify their software and envision how it is structured. Further, declarative representations

[1] The Swedish word *varv* carries the meanings "revolution" or "in layers." In geology, a *varv* refers to the annual sedimentary layer in a glacial lake. In the same vein, application code can be layered in our Varv system.

can facilitate future research on more usable, higher-level systems for reprogrammable interactive software.

2 THE VARV LANGUAGE

2.1 Design Goals

Varv's design is motivated by the following three design goals:

Provide a Structured Declarative Representation. Declarative representations have become widely adopted in various domains because they allow users to focus primarily on composing domain-specific primitives at a higher level of abstraction while deferring execution concerns to the underlying architecture or runtime [28]. Varv uses declarative language constructs to define application state and state transitions and provides an execution engine that parses declarative Varv specifications to produce an interactive application. Moreover, inspired by declarative representations of interactive visualizations like Vega [66] and Vega-Lite [65], Varv embeds its declarative representation of interactive software as a data structure expressed as JSON. In doing so, Varv lowers the threshold for programmatically reasoning about the semantics of interactive software. As a result, it becomes more feasible for the Varv architecture to hot-swap declarative specifications to enable live programming and an ecosystem of higher-level development tools to flourish (akin to the one found around Vega and Vega-Lite [70, 72, 73]).

Accretive Extensibility. Developers currently rely on extensibility APIs written by software creators to extend interactive software. However, such an approach presents a catch-22: it can be difficult, if not impossible, to customize an application in a particular way if the creator did not design a corresponding API. In contrast, to reduce a creator's burden of explicitly designing for extensibility, Varv defines interactive applications in terms of individual units called *concepts*. To extend a Varv application, a developer need only add a new specification to the runtime with entries to augment or override properties of existing concepts or introduce new concepts from a combination of existing parts. Thus, the extensions are themselves units that layer on top of the base definition of an application. This incremental approach facilitates experimentation: a developer can safely try implementing new features, or an end-user can selectively enable or disable extensions without fear of breaking or changing the original program. Moreover, this process of extension-by-addition simplifies resolving conflicting extensions by adding another specification to resolve the conflicting properties. Hence, our aim is the open authorial principle [7] that states that program modification should be possible purely through composition without rewriting existing code.

Decouple Application Logic from Interaction Modality. Existing methods for specifying interactive behaviors — namely, event callbacks — tightly couple an input event (e.g., mouse clicks, keypresses, swipe gestures) with the resultant action it triggers (e.g., selecting a piece on a board game, moving it from one square to another). As a result, retargeting an interactive application from one modality to another (e.g., desktop to mobile) or supporting custom interactive triggers (e.g., keyboard shortcuts) requires significant manual development effort. Varv decouples these two pieces: an application can be defined in an abstract, purely self-contained manner with custom, semantically-meaningful event names taking the place of low-level input events (e.g., `pieceSelected` instead of `click`). A subsequent specification then makes this abstract definition more concrete by binding semantic events to a specific interaction modality (e.g., `pieceSelected` is triggered by a `tap`).

2.2 Language Primitives

Concepts (see Figure 2 C) are Varv's core building block. They define individual named units of interactive behavior — for example, an "item" in a todo list that can be assigned or marked as completed, or a "piece" that can be moved along the squares of a board game or jump over other pieces. Each concept comprises a *schema* S, which determines the concept's state (i.e., data), and *actions* A, which enumerate the ways this state can change through interaction. Concepts can be augmented or extended in two ways: additional specifications can be introduced (e.g., Figure 2b) which reference an existing concept by name, and extend or override its properties; or, a variety of *extension* E operators can be used to define new concepts from existing parts.

Varv's concepts combine ideas from several different programming paradigms. At first glance, concepts seemingly map to classes in object-oriented programming (OOP), offering a mechanism for modularity, reuse, mixins, and traits. However, Varv's concepts make a fundamental departure: concepts are not encapsulated units (i.e., a concept's state and actions can be referenced from another). This design choice emulates the Store design pattern adopted by many popular JavaScript frontend libraries (e.g., Redux, Vue, and Svelte). Stores centralize application states, decoupling them from state transitions to aid rapid prototyping and developing cross-cutting components. Varv's unencapsulated concepts retain this affordance without sacrificing the modularity of OOP classes. We elaborate on these and other differences between Varv and existing programming paradigms in subsection 7.3.

2.2.1 Schema. The schema defines the shape and type of data associated with a concept. The syntax for the schema definition uses a modified version of JSON Schema [62] and supports a subset of the JSON Schema functionality. Varv extends JSON Schema with shorthands, making the language more concise and easier to read and write. For example, `{"label":{"type":"string"}}` can be written as the shorthand `{"label":"string"}`. Varv concepts can be referenced directly by name within the schema to specify nested state. For example, Figure 2 defines the schema of a `"todoList"` concept as an array of `"todo"` concept instances. Varv also supports deriving properties from existing state by specifying a `"derive"` object which expects an array of `"properties"` that are processed through an array of `"transform"` actions. Varv merges the properties and actions to generate a function that produces the derived value. For instance, in Figure 2, the `"totalCount"` property of the `"todoList"` concept is calculated as the `"length"` (a built-in action) of the `"todos"` property.

Early prototypes of Varv did not provide an explicit definition of concept state. Instead, the state was implicitly created and manipulated through sequences of actions. However, as we built increasingly complex applications, we discovered that this implicit treatment reduced *visibility* [12] into concept state (i.e., it was not

```
C  concepts: {
C    todoList: {
S      schema: {
S        todos: { array: "todo" },
S        completedCount: "number",
S        totalCount: { "number": {
S          derive: {
S            properties: [ "todos" ],
S            transform: [{ length: "todos" }]
S          }
S        }}
S      },
A      actions: {
A        updateCompletedCount: {
W          when: [{ action: "toggleCompleted" }}],
T          then: [...]
A        }
A      }
C    },
C    todo: {
S      schema: { text: "string", completed: "boolean" },
A      actions: {
A        toggleCompleted: {
T          then: [...]
A        }
A      }
C    },
C    assignable: {
S      schema: { assignedTo: "string" }
C    }
C  }
E  extensions: [
E    { join: [ "todo", "assignable" ],
E      as: "assignableTodo"
E    }
E  ]
```

(a) A concept definition that is abstract as it does not reference specific interaction modalities.

```
C  concepts: {
C    todo: {
A      actions: {
A        toggleCompleted: {
W          when: [{ click: { view: todoCheckbox }}]
A        }
A      }
C    }
C  }
```

(b) Extending the abstract specification with concrete references to modality-specific input events (the `toggleCompleted` semantic event, defined in the abstract concept, is triggered when the `todoCheckbox` widget is clicked).

Figure 2: The components of a Varv concept definition for a simple todo list. As a convention, and to demonstrate the merging of concept definitions, we split the definition into an abstract and a concrete part. The abstract part provides definitions for a `todoList`, a `todo`, and an `assignable` *concept* C. Each concept has a *schema* S and the `todo` concept has an *action* A which encodes a state transition (omitted) in a *then-block* T. An *extension* E is used to create an assignable todo by joining the `todo` and `assignable` concepts. The concrete part binds the `toggleCompleted` action to an interaction specific to a DOM view using a *when-block* W. (Quotation marks from JSON keys removed for readability.)

clear what properties were available for access on a given concept). In contrast, by explicitly enumerating a concept's properties and their types, Varv schemas help formalize concept state. They serve as a baseline level of documentation for the structure of concepts within the program, and types are validated at runtime to reduce *error-proneness* [12]. Schemas, moreover, aid concept reusability. For instance, in early prototypes, Varv stored concept state directly on DOM nodes. This approach introduced *hidden dependencies* [12], making it challenging to adapt concepts to new contexts without introducing knock-on effects to the output interface. It, similarly, introduced a *premature commitment* [12] by requiring every concept to be reified as an interface element. In contrast, with schemas, concepts can be reasoned about in purely abstract ways and referenced throughout a declarative specification without being mapped to a concrete user interface component.

2.2.2 Actions and Triggers. Actions provide a common abstraction for specifying state transformations, and consist of two parts: an optional *when-block* W and, a required *then-block* T.

The when-block defines an array of triggers or events that cause the action to be executed. Varv provides two types of triggers (see Appendix C). Reactive triggers govern concept space: they fire when a concept's state changes, or when a concept's action finishes executing, or at a given interval. For instance, in Figure 2a, the `updateCompleteCount` action makes use of a reactive trigger — this action executes once the `toggleCompleted` action of the `todo` concept has run to completion. View triggers, on the other hand, fire when input events (e.g., mouse clicks or key presses) occur. For example, Figure 2b demonstrates how an additional specification can bind purely abstract concrete definitions to concrete interface elements using view triggers — the `toggleCompleted` action of the `todo` concept fires when the `todoCheckbox` element is clicked.

The then-block specifies an array of actions that should be executed. Nested actions can include either other concept actions or Varv's primitive low-level actions (see Appendix B). These built-in actions include operations for manipulating a concept's state (e.g., arithmetic calculations, string and array manipulations, etc.) as well as determining control flow (e.g., early exiting a chain of actions, or forking the chain to execute an independent action). This design allows for recursion (i.e., an action can call itself within the then-block), with a `"where"` control flow action used to indicate the terminating condition. The output of an action can be referenced using the dollar sign — by default, the output is named for the action (e.g., `$length` references the output of an upstream `"length"` action) but these variables can be renamed using the `"as"` property offered on many actions. Finally, actions can be parameterized using the using the `@`-symbol in front of parameter names, e.g., `"@newTodoLabel"`. These parameters can subsequently be provided as properties when referencing the action downstream. The `addNewTodo` action shown in Appendix A.1 provides a complete example of these ideas. When it is executed, it creates a `"new"` instance of the `todo` concept using the value provided by the `newTodoLabel` parameter (populated on line 45). The output of this action is stored in the `$newTodo` variable (due to the `"as"` property specified on line 25), and is used to append to the list of todos.

Concept actions do not need to define both blocks. Rather, concept actions can be directly defined as a then-block (bypassing

the nested format) and additional, separate specifications can later bind actions to specific interface elements. For example, Figure 2 uses this convention to first define the abstract idea of a `todoList` comprised of `todos` which can be `completed` (Figure 2a). Note, the `toggleCompleted` action does not define a when-block. In a subsequent specification (Figure 2b), this action is bound to `click` events that occur on the `todoCheckbox` element. By following this convention, a concept action can serve as an abstraction for a sequence of nested actions, and helps decouple the application logic of an interactive component from a specific reification or modality.

Varv also allows users to register custom actions written in JavaScript (see Appendix A.4). Custom actions can extend the Varv standard library with additional functionality, integrate Varv with existing JavaScript code, or let users write complex business logic using imperative code.

2.2.3 Extensions. Extensions are mechanisms that enable the reuse of concepts. Out of the box, Varv supports merging and overwriting properties using naive declaration merging based on JSON keys. Figure 2 uses declaration merging to extend the `toggleCompleted` action on the todo concept with a when-block. However, during our prototyping process, we quickly realized that naive declaration merging is limited to only extending or overwriting existing concepts. In particular, there is no way to use naive declaration merging to build higher-level concepts that are ad hoc compositions of existing concepts.

To support more nuanced mechanisms for concept reuse, Varv offers four extension operators: `"inject"`, `"join"`, `"omit"`, and `"pick"`. `"inject"` merges the definition of one or more source concepts into another target concept. The source concepts are left unaltered while the target concept gains new functionality. The `"join"` operator is similar to `"inject"` but merges one or more source concepts to create a new concept, leaving source concepts unaltered. The `"omit"` operator takes a source concept and can remove actions and schema from the concept, altering the source concept, providing a mechanism to remove functionality via addition. The `"pick"` operator takes a source concept and selects a subset of the schema and actions to create a new target concept, leaving the source concept unaltered. Using these four operators, users can define a library of concepts as mixins and inject them into other concepts to prototype applications rapidly.

2.3 Event Flow

Varv is an reactive and event-based system. Events in Varv are data objects that are used to transfer information. Events are emitted from triggers, passed on to actions, and then terminate once an action is performed.

2.3.1 Event Contexts. Events contain *contexts* and *shared variables* (see Figure 3). A context is also a data object that consists of a concept instance, the *target*, and variables in the context. The target is required by many actions to define on which concept instance an action should work. For example, consider a `todo` concept that contains the string property `text`. The action `{"length":"text"}` computes the length of the `text` property. In order to know from which instance the action should take the `text` property from, the target is used. Once the action is performed, the `"length"` action adds the variable `length` with the result to the context variables of the respective context.

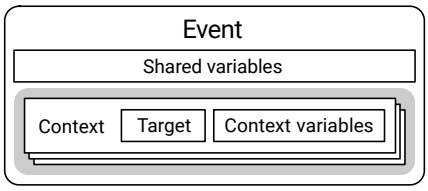


Figure 3: The structure of an event in Varv.

An event can contain multiple contexts, because actions might need to work on multiple instances at once and do something for each of them. This is inspired by JavaScript array methods such as `map` [53]. The contexts of an event can be modified by actions, e.g., the `"select"` action replaces the current contexts in an event with one context for each concept instance the selection defines. Other actions also enable to remove contexts from an event, e.g., the `"where"` action filters contexts based on the properties of targets or variables in the context. Figure 4 shows an example where first the `"select"` action is used to select all `todo` concept instances, then the `"length"` action is used to retrieve the length of the `text` property of a `todo`, and lastly the `"where"` action is used to filter the one with a length of less than four characters.

2.3.2 Event Creation and Passing. Events are created by triggers. When creating an event, a trigger can add contexts to a new event, for example, the `"click"` trigger adds the concept instance of the element the user clicked on — if it is a concept instance — as a target and the coordinates of the mouse click as variables.

Events are by default passed from one action to the next, each working on the same event. This, however, can lead to changes to the variables or contexts of an event. If an action should be performed without affecting the event, the `"run"` action can be used. This effectively makes it possible to split the event up. If an action removes all context from an event, by default, an empty event without contexts is passed on to ensure the execution of consecutive actions. In this case, however, the event would lose all its context variables. To prevent this, Varv stores variables that are the same across all contexts in the shared variables. These are persisted even if no contexts are in the event anymore and added back to context variables once new contexts appear. If an event should not be passed on if there are no contexts left, actions like `"select"` or `"where"` have an option to stop the event, allowing them to act as a gate.

3 THE VARV ARCHITECTURE

The overall architecture of Varv consists of six main components: the *event engine* that reads in *concept definitions*, *templates* that define how these concepts are rendered in the *view layer*, and *mappings* that define where data from concept instances should be stored in the *data layer* (see Figure 5). This section summarizes the purpose of each of these components; their implementation in our Varv prototype is described in section 4.

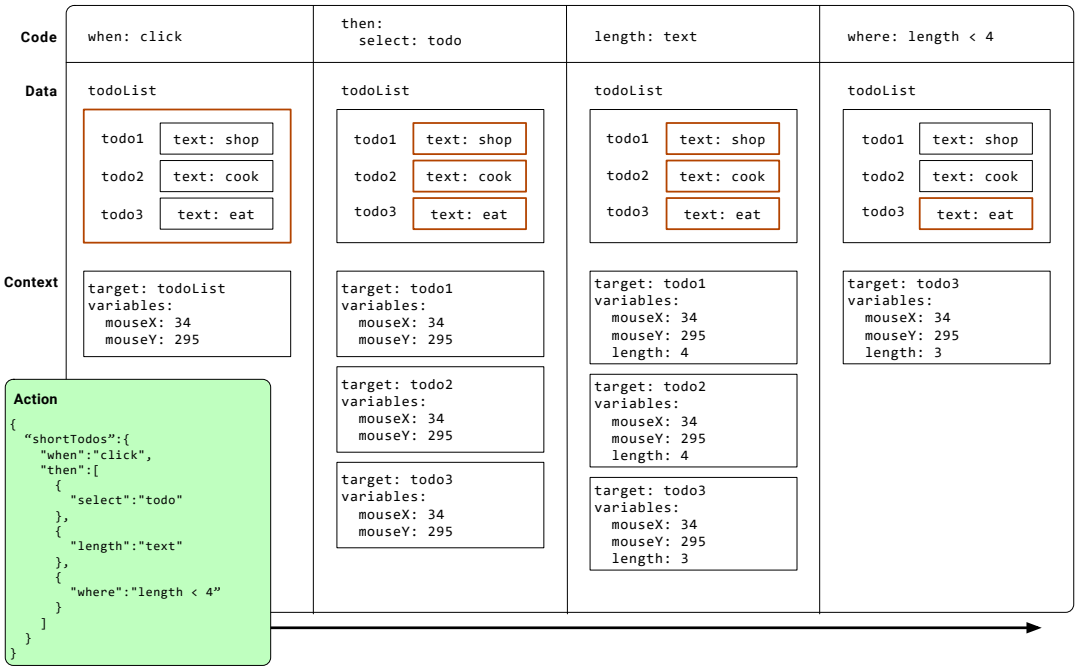


Figure 4: Example of an event flow in Varv. The user clicks on the todo list in the app, triggering the action. The syntax of the `"where"` action was shortened here to simplify the example.

3.1 Concept Definitions and the Event Engine

Concept definitions are files that use the concept language which was introduced in subsection 2.2 to define the interactive behavior of an application. There can be any number of concept definition files in a Varv application. All concept definitions are merged by the event engine at runtime. When being merged, concept definitions later in the document overwrite earlier ones — i.e., existing concept, actions, and properties can be added, suppressed, or overwritten by adding new concept definitions at the end of a document.

To illustrate this merging process, we used the convention of splitting concepts into two parts in our examples: an abstract part, that contains actions that are view-agnostic, and a concrete part that contains actions that are view-dependent. By separating these parts, it is possible to reuse the core logic of a concept if another view is targeted.

3.2 Templates and the View Layer

The view layer contains *views* and *templates*. A view is a component that renders a user interface with which users can interact, for example the DOM. Making the view independent from the event engine, allows it to connect different types of views to the same underlying interactive behavior and state of an application.

Templates are used to specify how state should be represented in the view by referring to concepts and properties in them (see Appendix A.3 for an example). A template is view-dependent, thus, different views require different templates. In the DOM, for example, a template could be written in HTML while in other views they might be required to provide a scene graph or other structures. The view then combines these templates with the state it retrieves from the event engine to generate a user interface. By generating the user interface in this way, elements in the view can be connected to their underlying concepts and state, allowing for higher-level tooling such as a view inspector (see subsection 6.2). Lastly, views can also add view-dependent *view triggers*, which can be used in actions in the concept definitions to react to user input in the view.

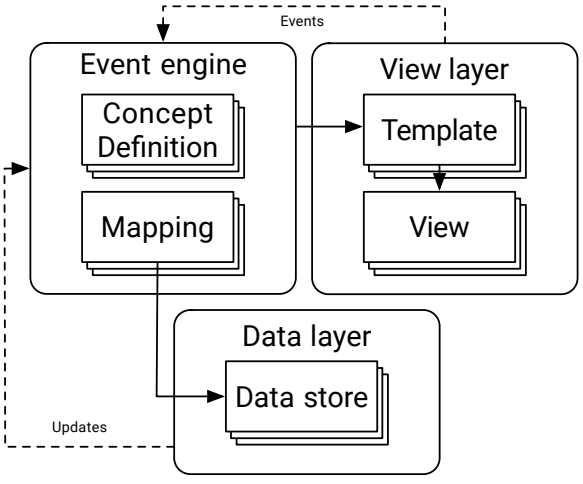


Figure 5: Architecture overview of Varv.

3.3 Mappings and the Data Layer

The data layer contains *data stores*. Data stores allow Varv to store state of concept instances and their properties in them. A data store can be anything that can store data in a key-value format. One purpose of using a separate data layer in Varv is to be able to dynamically store data in heterogeneous ways, which allows users to define properties in the schema of concept definitions without having to take care of how and where it is stored. Another reason for using a data layer is to decouple the state of an application from the interactive behavior. This, for instance, enables to hot swap concept definitions in the event engine or to connect different application to the same data store. The latter allows users to create their own personalized applications, but still being able to collaborate on shared data (subsection 5.1 demonstrates this).

Mappings are pointers that define in which data store state is stored. Mappings can be defined for each property of a concept. This allows, for example, to store ephemeral state like the content of an input field in a data store that is not shared with other users. If properties are mapped to multiple data stores, Varv synchronizes state between all selected data stores. Data stores can, further, notify the event engine about updates to the data, for example, if a remote user changes data in a shared data store. The event engine then synchronizes the data with other data stores and notifies actions and the view about the change.

4 IMPLEMENTATION

Our main implementation of Varv[2] is written in JavaScript, builds on top of the Webstrates [41] platform and the Codestrates v2 [13] framework, and runs purely client-side in a Web browser and uses Codestrates v2's extensible in-app IDE Cauldron for development (see Figure 6a). This section will first describe the Webstrates platform and Codestrates framework and what parts are used for Varv. Then we explain how the control flow of Varv works and how we achieve live extensibility.

We have also implemented a proof-of-concept version of Varv that is independent of Webstrates. We use this version of Varv to package Varv applications as Electron [60] apps using regular JSON and HTML files stored on the disk for concept definitions and templates (see Figure 6b).

We, additionally ported this version of Varv to Observable [57] using tagged templates [54] for concept definitions and templates. This makes it possible to use the computational notebook view of Observable to create, share, and incrementally develop Varv applications (see Figure 6c). Further, this demonstrates the portability of the Varv runtime to contexts outside of Webstrates.

4.1 Building on Webstrates, Codestrates v2, and Cauldron

4.1.1 Webstrates. Webstrates [41] is a software platform for building reprogrammable, collaborative software on the Web purely from the client side. The simple yet powerful mechanism behind Webstrates is to synchronize and persist changes to the DOM of a web page served from the Webstrates server. This includes changes to embedded code (JavaScript, CSS, and more), effectively making it possible to both collaborate on *using* and *programming* software. As default the whole DOM is synchronized, but to support a relaxed WYSIWIS (What You See Is What I See), a custom `<transient>` element can be used to create subtrees that are not synchronized — e.g., for UI elements.

4.1.2 Codestrates v2 and Cauldron. Codestrates v2 [13] provides a model for controlling the execution and interdependence of scripts of various types.[3] Furthermore, it provides an API for instantiating code editors for specific scripts (stored in so-called *code fragments*) in the user interface. Codestrates v2 is bundled with its own extensible development environment *Cauldron*, which allows *within-application modification*: users are able to create, edit, and run code fragments directly inside the web browser without additional software (see Figure 6a). Codestrates v2's execution engine can be used independently of Webstrates (e.g., as in our Electron prototype).

4.1.3 Varv. Varv adds a new Codestrates v2 fragment type for concept definitions. Templates are stored in HTML fragments and styling in CSS fragments. Varv leverages the synchronization with the Webstrates server to synchronize state that is stored in the `"dom"` data storage — enabling collaboration. Varv inherits the ability to edit code directly in the interface, collaborate in real-time, and version both data and code from Webstrates. Concept definitions, templates, and the concept data store are all persisted in the DOM in custom tags hidden from the browser view using CSS. The user interface generated from Varv is wrapped in a transient element, hence synchronization of application state only happens through the data storage.

4.2 Event Engine

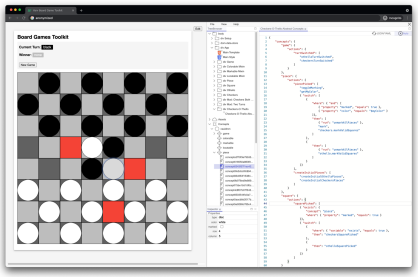
4.2.1 Building and Rebuilding the Model. The event engine queries all concept definition fragments and parses their JSON code. Concept definitions are merged sequentially into a single definition. Extensions to the concepts such as injections are performed after the merge in the order they appear in the concept definitions. When a new concept definition is added or any of the existing ones are changed or deleted, the running model is destroyed and a new model is built. Application state is not lost as it is stored separately in data stores.

The merged model contains all concepts, schema, actions, mappings of properties, and data stores defined in the concept files. Once merged, the engine uses the mapping and data store information to connect properties of concepts to their mapped data stores and notifies the data stores of their connection. Afterwards, the view is notified of the updated model. Lastly, the engine subscribes each action that has a when-block to their respective triggers.

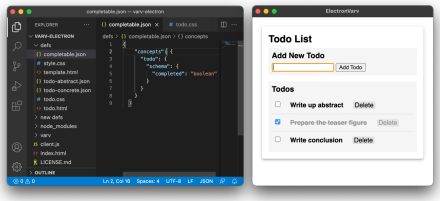
Primitive triggers register themselves in the event engine once instantiated. Once the trigger fires an event, which consists of a string containing the trigger name and a JavaScript object containing the context, the event is passed to the event engine that distributes the event to the actions subscribing to that trigger. Like triggers, primitive actions register themselves in the event engine

[2]Varv on GitHub: https://github.com/Webstrates/Varv (Retrieved November 25, 2021)

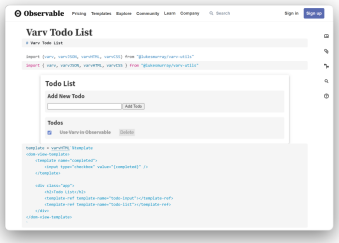
[3]While this model enables executing JavaScript code at runtime, Codestrates v2 does neither handle duplicate event listeners, other issues that come up when re-executing imperative code at runtime, nor synchronize runtimes across clients. Hence, limiting its use for live and collaborative programming.



(a) The main implementation of Varv. It builds on top of Webstrates. The Cauldron editor can be opened in the web browser.



(b) A proof-of-concept implementation of Varv in Electron.



(c) A proof-of-concept implementation of Varv in Observable.

Figure 6: Screenshots of our implementations of Varv.

once instantiated. Once an action is triggered, its actions are executed: Each action receives the list of contexts in the event and the action options defined in the concept definition (see subsection 2.3 for more detail on the event flow).

Actions with the same name can be defined in multiple concepts, thus, we provide a look-up function to find the correct action. To target a specific action implemented in a concept, a dot-notation can be used, for example, `"checkers.markValidSquares"` or `"othello.markValidSquares"`. There is a lookup order starting first with primitive actions to searching for actions with a given name in any concepts in the model.

4.3 Data Stores

Types of data stores are registered in the event engine like triggers or actions. They can be used to create custom named data stores in concept files. Our implementation of Varv defines three types of data stores: `"dom"`, `"localStorage"`, and `"memory"`. By default, properties are mapped to the `"dom"` data store, where they are persisted and synchronized with other clients through Webstrates. An option for the `"dom"` data store can change the location for storing the state in the DOM to another webstrate, allowing multiple applications to work with the same data. Properties can also be stored in `"localStorage"` or `"memory"` data stores, if they are ephemeral or should not be shared with other clients. Our Electron-based prototype uses the `"localStorage"` data store for persistence.

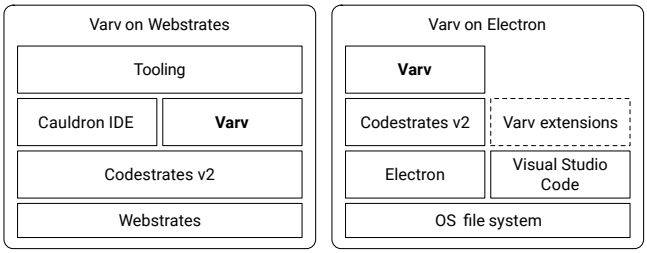


Figure 7: The software stack of two of our Varv prototypes. Our main Webstrates-based implementation uses Cauldron as its editing environment with Varv-specific tooling built on top. The Electron-based prototype uses Codestrates for code execution but is independent from Webstrates. Electron is used to store and load code from the file system. Code can, e.g., be edited using Visual Studio Code, which could be extended (not implemented in our prototype) with Varv support by using JSON Schema or by porting our block-based editor.

Once the event engine has loaded the model, it connects itself to the data stores defined in the model. Next, it maps each property to the data stores that it is mapped to and registers "getter" and "setter" callbacks of the data stores in the property. After registering the callbacks for each data store, the event engine attempts to load already existing data of a property from each data store and publishes it to all other data stores of that property and, hence, synchronizes them. If a property is mapped to multiple data stores that contain conflicting data, the ones first in the list of mappings overwrite the data of later ones.

If data changes outside the Varv system, e.g., remote changes to the `"dom"` data store, a data store can notify the event engine about changes to properties, which will then synchronize it with other data stores and notify views and the `"stateChanged"` trigger. Changes to properties from actions or views are sent to the event engine and forwarded to the registered data stores.

4.4 Views

Views exist mostly independent from the rest of the Varv system. They can connect to the event engine and register to concepts and properties to get and set properties, as well as register their own triggers. Our implementation includes the `"dom"` view that renders data in the DOM of a website.

It parses all `<dom-view-template>` nodes and collects what concept instances are required by the templates. Next, it subscribes to these concept instances in the event engine and retrieves a list of concept instance objects with references to their properties. The view is notified on updated properties, created or removed concept instances, and is able to set changes of properties — again, passing new values to the event engine, which forwards it to the data stores. If any template changes, the view unsubscribes all properties and repeats the process. In the templates, the `"dom"` view looks for special attributes (`concept`, `property`, and `value`) and replaces curly braces of properties in other attributes or text nodes with the

values of properties (see Appendix A.3). Additional style can be added using CSS and assets like images or icons can be uploaded to a webstrate using Cauldron. The `"dom"` view supports mouse and keyboard events as view triggers.

5 CASE STUDIES

To illustrate how Varv applications enable new ways of extending and modifying software, we present two case studies of how Varv can be used: The first case study illustrates how an existing todo list application can be collaboratively modified through addition of code. The second case study demonstrates how Varv can be used to create a declarative abstraction layer for board games and how different applications can be built using these abstractions. The case studies are also presented in the paper's accompanying video. In addition to the two case studies, we briefly describe other application examples we explored.

5.1 Case Study 1: Todo List

Imagine two computer science students, Melissa and Daniel, who work together on a course project. To manage their tasks, they use a simple collaborative todo list web-app created in Varv. During the first half of their project they work tightly together. However, they increasingly need to split up tasks and work on them in parallel. They now want a feature in the todo list that lets them assign todo items between them.

Adding the "assignee" field. To modify the todo list they click an "Edit" button in the top right corner of the interface to open Cauldron. There is a list of files that includes the concept definitions for the app: `"todo"`, `"todoList"`, and `"todoInput"`. They need to modify the `"todo"` concept. Daniel creates a new folder with the name "assignee" with a new concept file. He adds the new property `{"assignee":"string"}`, which stores the name of the person responsible for the todo item (see Figure 8a). Next, he needs to show that information in the view, so he also creates a new template file. He copies over the template from the original todo item and adds a line with an input field for the property. While doing so, he can immediately see the input field appear in his view and test if it works by writing his name into the input field.

Adding filtering. Melissa's friend Samantha has written a filtering mechanism for the todo list. Melissa can add the filtering to their app by dragging the folder containing a concept, template and style file, from Samantha's in-app IDE Cauldron to hers. Daniel wants filtering on his own `"assignee"` field as well. However, he does not really understand the filtering code so he asks Melissa for help. Together, they try to understand the code and Melissa adds some code to the filtering to also support the assignee filtering. While Melissa is coding, Daniel has the app open, and he can immediately try out the effects of code on the app.

Using separate views. During the second half of their course, Daniel adds more and more features to the todo list. Melissa finds the interface cluttered and wants a simpler app. So, she creates a copy of their todo list. In the copy, she deactivates all the features she does not want in her version of the todo list and — because she is making changes anyway — also adds a dark theme for the web-app. To still be able to work on the same data as Daniel, she remaps the data store of the new app to Daniel's (see Figure 8b).

How it works. Varv supports incremental application development, thus, Daniel and Melissa can add functionality step-by-step. Adding new concepts or templates allows them to overwrite the parts of an app that they want, without having to change the original implementation. This is enabled by the event engine merging all concept definitions and rebuilding the model after every change. This, further, enables them to add new functionality from their peer Samantha without having to touch the code of the original todo list or their assignee feature. As they are adding new functionality accretively in new concept definitions, it is also possible to go back to prior versions by disabling these definitions in Cauldron.

With Varv running on Webstrates, they can collaborate on the code of the app in Cauldron and test new functionality together. Varv makes the collaborative testing possible by automatically reloading concept definitions whenever changes are done locally or remotely. As the todo app is stored in a webstrate where the app is self-contained, i.e. both the data and the application code are stored together, they can generate copies to create personalized applications. Decoupling the interactive behavior (concept definitions) and the view from the data, further, makes it possible to remap the `"dom"` data store to another webstrate, providing means to create customized views while using the same data.

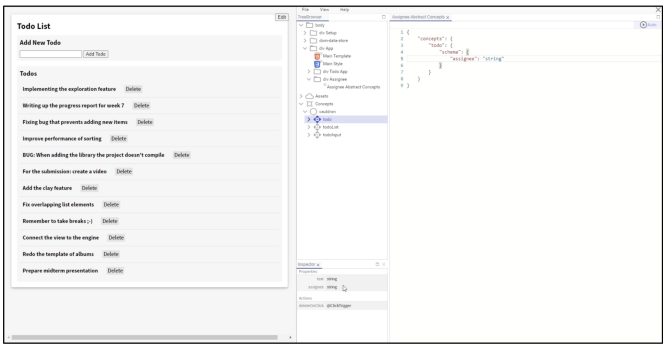
5.2 Case Study 2: Board Game Toolkit

Sean is a fan of board games like Checkers[4] or Othello[5]. He has ideas for modifying existing games to make them more attractive and wants to realize some of them as web apps to play with friends.

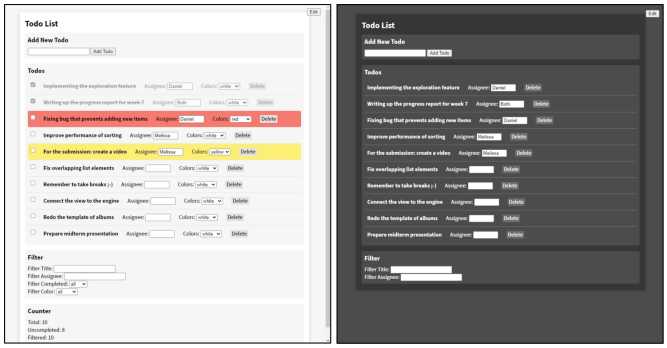
Building a toolkit. Sean wants to make a toolkit for games in Varv so he does not have to build new games from scratch. He starts by creating a `"game"` concept for more general game mechanics like taking turns and who the winner of a game is. Next, he creates the basic concepts of board games: the `"piece"` and the `"square"`. Both share common traits: they have one of two colors and a location defined by a row and a column. Sean creates a shared mixin with helper actions for each of those and calls them `"colorable"` and `"locatable"`. He injects both mixins into both concepts. In order to let players select and move pieces, he adds another mixin that he calls `"markable"`, which enables him to mark pieces or squares. The mixin contains actions to, for instance, `"mark"` a piece or check whether a piece `"isMarked"`.

Creating Checkers and Othello with the toolkit. Next, Sean adds a new concept for the checkers game, where he adds actions that are specific to Checkers, for example, to handle when a piece jumps over another piece. In doing so, he uses the actions from the concepts he created in the toolkit. Sean shows the game to his friend Amy. She wants to implement a game by her own. Amy makes a copy of Sean's game and disables the Checkers concept file. She then creates a new concept file for her favorite board game Othello. After implementing the Othello game, she immediately tries out the game in a online multiplayer match against Sean.

[4] Checkers or Draughts: https://en.wikipedia.org/wiki/Draughts (Retrieved November 25, 2021)
[5] Othello/Reversi: https://en.wikipedia.org/wiki/Reversi (Retrieved November 25, 2021)



(a) Daniel adds a new concept definition to the todo list app and adds the "assignee" property. Once the concept definition is activated, the property is immediately added to todos, seen in the inspector in the center bottom of the screen.



(b) Daniel and Melissa can both use their preferred view and functionality in their app. While Daniel (left) uses more features and a light theme, Melissa (right) uses a more simple layout with a dark theme. The underlying data is shared.

Figure 8: Screenshots of the first case study.

Modifying the games. Sean plays around with variants of his Checkers game. He makes concept definitions that he can toggle on and off with small adjustments, which, e.g., enables pieces to move both forward and backwards all the time — making the game more complex to play. In yet another variant, he lets players have two consecutive turns after each other. For a final variant, Sean wants to combine the game rules of Amy's Othello game with Checkers. He asks Amy to join him remotely in creating their own "Frankenstein-game" *Checkers-O-Thello*. They add a new concept definition, where they resolve issues between the game rules of both games. After some fixes, they activate the game rules of both Othello and Checkers and can now use Othello's game rules for placing pieces and Checkers' game rules for moving pieces.

How it works. Varv lets users create their own abstractions over complex state transformations in the form of custom concepts and actions. Sean leverages this by creating concepts for pieces and squares and by adding meaningful actions to them. By doing this, he effectively writes his own domain-specific language for creating board games. Using this language in the Checkers game, he can think about high-level rules of the game, such as "Which are the valid squares a piece can move to if it was selected?" rather than low-level problems like "How do I detect if the when the user picks a piece?" Once created, these abstractions can be reused, so that, for example, Amy can also create her Othello game without having to solve low-level problems first.

This process of modifying games and creating variants of them is supported by Varv's support for incremental application development. It enables Sean to modify only some actions of the Checkers game in his variants, without having to recreate the whole game several times. When implementing the *Checkers-O-Thello* game, Varv's real-time collaboration makes it possible for Sean and Amy to work together on the code, and state synchronization through the Webstrates-based data storage to play the game as a multiplayer game. As both of their games were created using the same abstractions, merging them is a straightforward task. They need to add a few actions to their game implementations to get the game rules of both games work together in a single game. By accretively adding these actions, they do not even have to touch the concept definitions of the two already existing games — something that would be difficult to do in conventional imperative programming languages.

5.3 Other Examples

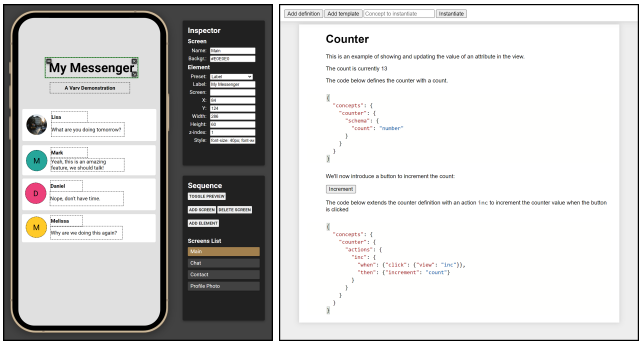
Besides the two case studies, we also explored creating other types of applications with Varv:

UI Designer. The UI Designer can be used to create mock-ups of user interfaces and the navigation of apps — similar to Figma [22]. It lets users create multiple screens and add elements such as labels, boxes, or buttons within those screens (see Figure 9a). Elements can be moved and resized with the mouse cursor and can link to other screens. In the preview mode, interactions can be tested and used to navigate to other screens by clicking on them.

Computational Notebooks. The Computational Notebook is written in Varv and lets users write their own Varv applications using a computational notebook interface. Each cell in the notebook can be a concept definition or a template making it possible to quickly sketch Varv applications (see Figure 9b). New cells can be added using buttons in the toolbar. The Computational Notebook also adds a custom action `AddFragment` (similar to Appendix A.4), written in JavaScript, which can add new fragments to the DOM.

6 TOOLING

To demonstrate how the architecture of Varv and its declarative language design lends itself to create tooling on top of it, we implemented multiple authoring and debugging tools for Varv. The JSON-based data structures, in which Varv applications are defined, are simple and structured, so other authoring environments can be used to author Varv applications. The decoupled architecture of view, data, and event engine, in addition, facilitate to create inspectors for data and the elements in the view — enabling not only to inspect the view of applications but also their interactive behavior.



(a) The UI Designer. **(b) The Computational Notebook.**

Figure 9: Screenshots of other example applications we explored with Varv.

6.1 Authoring Tools

By specifying applications in Varv in JSON-based data structures, a common file format in the modern Web, Varv provides a common interface for other authoring tools to connect with. We created three examples of authoring experiences that allow to define interactive behavior in Varv.

YAML-Based Editor. Readability and ease of writing actions could be improved by using YAML instead of JSON as the language for concept definitions. As a superset of JSON, it is possible to add support for specifying concept definitions in YAML instead of JSON. Using indented delimiting, YAML potentially makes writing code easier as less special characters are used (see Figure 10a).

JSON Schema Auto-Completion. We created a JSON Schema [62] specification for the concept language and most of its primitive actions and triggers, and data stores. JSON Schema is widely used and supported by many code editors and IDEs as well as Cauldron. Registering the JSON Schema in these editors enables autocompletion, type checking, and validation (see Figure 10b). Autocompletion supports users in exploration, while type checking and validation can help to resolve wrong specifications/parameters while writing the code. A current limitation of the JSON Schema is that it is limited to the primitive actions and triggers, actions that are defined in concepts are currently not added.

Structured Block-Based Editor. To create a more tangible and explorable authoring experience, we implemented a structured and block-based editor (see Figure 10c). The editor is implemented using the Blockly [26] library and provides blocks for most primitive actions and triggers. The sidebar of the editor makes it easy to explore available actions and triggers. Editing concept definitions in the editor automatically updates the JSON, creating a live programming experience. By applying changes immediately to the JSON and hence the event engine, the editor allows for quicker ways to enable and disable actions and experimenting with different interactive behavior. The block-based editor, however, has the same limitation as the JSON Schema, as it currently not dynamically adds actions from concepts as blocks.

6.2 Debugging Tools

The declarative structure of applications and the decoupling of the engine from the data and the view layer means that the view is generated from the data and the model in the event engine. In doing so, the view can be connected to both the interactive behavior and the underlying data. We show in two inspection tools how this connection can be leveraged to support debugging and testing. By bringing applications and the development environment with their underlying code closer together, we aim to make it easier for users to find the relevant code for their planned modifications, lowering the threshold to modify their applications.

Data Inspector. The data inspector lives inside the Cauldron editor (see Figure 11a). In its tree browser, concepts types and their instances can be modified, created, or deleted. Selecting a concept shows its schema and actions, and selecting a concept instance shows its properties and their values in the inspector tab underneath the tree browser. Values can be edited and modifications are directly applied in the view. Creating the data inspector was possible as the information about schema and actions of concepts is available in the model of the event engine in a structured format.

View Inspector. The view inspector can be used in the `"dom"` view. By holding the control key and right-clicking on any element in the view, the view inspector shows a menu with information about the clicked element (see Figure 11b). The view inspector checks if the selected element or any of its parent elements is an instance of a concept and which template files were used to generate the view. Using the information about the concept instance, the view inspector creates a link to the instance in the data inspector that users can follow to inspect the properties of the selected element.

The view inspector is enabled by the decoupling of concept definitions, data, and the view. As the view is generated at runtime and updated whenever a concept definition or template is modified, it always retains a connection to the concepts and data that were used to generate it. We created the view inspector as a step into breaking up the strict border between the application and the development environment, supporting users in finding relevant code for their modifications.

7 RELATED WORK

7.1 Declarative Programming

Declarative languages separate the *how* from the *what* and allow users to focus on the specifics of their domains [28]. Some would argue that Varv is not declarative because actions written in Varv consist of series of steps and can assign variables. However, computer science literature does not provide a concrete notion of what declarative programming is. Lampson describes a declarative program as a program which has *few steps*, is a *good match* for the users view of the problem domain, provides mechanisms for *composition*, gives *big primitives* so that users can get a lot done without having to write code, and allows for *clean escape hatches* so that imperative programming is allowed when needed [43, 63]. Varv meets all of these conditions. Varv provides high level primitives for binding data and updating state, enables rich mechanisms for composition, and allows users to specify custom actions using JavaScript. Additionally Varv provides capabilities for users to develop their own

(a) A YAML editor for concept definitions.

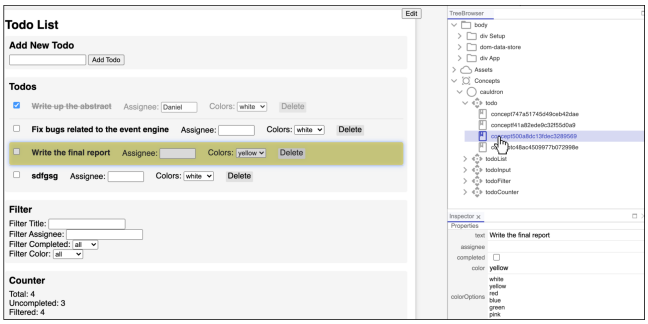
(b) JSON Schema for concept definitions adds autocompletion and code validation.

(c) A block-based editor that enables exploration of actions and triggers in the sidebar.

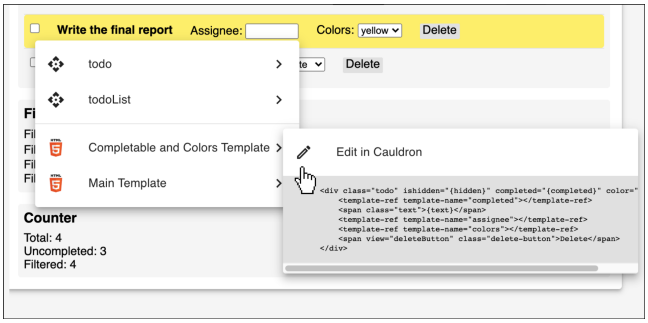
Figure 10: A summary of three implemented authoring tools in addition to editing JSON directly.

domain specific primitives, enabling greater expressivity in the large domain of interactive web applications.

Declarative languages have become widely adopted in many domains because they make it easier to accomplish complex tasks.



(a) The data inspector in the Cauldron editor. It can inspect concepts and their instances. The properties of instances can be edited inside the editor. Selecting a concept type or instance highlights the related element in the view.



(b) The view inspector in the "dom" view. It can inspect elements and identify their concept type. Using the menu, it is possible to jump to concept instances in the data inspector or to the template files used to generate the view.

Figure 11: Two debugging tools for inspection we implemented for Varv.

Database query languages such as SQL have allowed database developers to focus on describing what data they want while the query optimizer determine how best to get the data using available indexes and joins [40]. HTML and CSS let web developers describe what markup and styling to use while the browser optimizes the page rendering [28, 40]. Vega-Lite lets users describe high level incomplete visualization specifications and uses heuristics and rules to resolve ambiguities and generate a visual representation which follows visualization best practices [72]. Beyond performance improvements, declarative languages are highly suitable for integration with higher level tools [66]. Within the Vega and Vega-Lite ecosystem Voyager [72], Lyra [64], and Altair [70] have been developed to let users generate visualizations through exploration, direct manipulation, and Python bindings respectively.

Because of the benefits of declarative languages there have been many attempts to write declarative languages for the web. Many of these attempts such as Araneus [55], AutoWeb [25], STRUDEL [21], and WebML [15] provide declarative languages, both graphical and textual, which can be used to derive multi-page websites from various data sources. These projects use multiple approaches for specifying the structure, navigation, and presentation of websites, but are focused on sites where each page is a statically generated view of data rather than an interactive application. SOBL [19] is closer to Varv and provides a declarative specification for user interactions which is automatically parsed into static HTML web pages and state transition diagrams, but does not close the loop and generate interactive applications, or provide mechanisms for composition of existing programs.

Vega and Vega-Lite [65–67] provide mechanisms to allow users to convert a definition of a data visualization, written in JSON, into an interactive chart. Varv is an extension of the same idea to applications. Because of the similarity, Varv uses many similar mechanisms to Vega and Vega-Lite. For example both Vega and Varv allow users to define reusable pieces of functionality by associating the functionality with a name and both Vega and Varv allow users to extend the runtime with *user defined functions*, while still allowing users to invoke the functions declaratively.

KScript and KSWorld [58] are respectively a scripting language and an editor for end-user authoring of software, that emphasize reduction of accidental complexity and live programming to support

exploratory application building. Hence, the project shares similar goals with Varv. Also, similarly to Varv, KScript provides declarative language constructs for event-flow based programming. However, they are embedded in an object-oriented imperative programming language inspired by JavaScript, whereas Varv is a fully declarative programming model.

7.2 Alternative Representations of Web Applications

There is a long line of research which attempts to make writing simple web applications easier. Object Spreadsheets [50] identifies that many end-user programmers are familiar with the spreadsheet model, and uses a new computational model for spreadsheets to enable the development of web applications. However, Object Spreadsheets is focused on providing powerful spreadsheet based mechanisms for data modeling, and provides few abstractions for enabling interactivity, and falls back on imperative scripting for mutating state. Quilt [11] provides a similar spreadsheet backed metaphor for web applications, but provides almost no data abstractions, and acts essentially as an HTML attribute based template language for binding elements on a web page to rows in a spreadsheets. Gneiss [16] provides a live programming environment for developing websites from web data using spreadsheets but does not provide mechanisms for code reuse or composition. Wildcard [46, 47] also uses a spreadsheet metaphor, but enables the augmentation of existing websites with additional data rather than the construction of independent applications. Varv provides a declarative specification for application logic and user interactions, as well as data bindings to a data store. Because Varv represents a declarative target, we believe high-level tooling such as live editing environments or integration with external data sources could be built on top of Varv.

Mavo [71] allows users to develop CRUD applications with a template language built directly into HTML. The user defines a data schema *implicitly* by adding attributes and expressions to HTML elements, and Mavo provides out of the box support for editing data directly in the interface. The primary goal in Mavo is to allow users to directly manipulate and define the shape of a data schema in a UI layout. Varv also supports a template language but separate the definition of the data model from the template. Varv has less emphasis on direct manipulation of data and instead focuses on composition and malleability of concepts. In addition, by separating data from the view, Varv allows users to write application logic once while targeting multiple view layers. Additionally, because the data logic in Mavo is encoded directly in the layout, creating new layouts while retaining existing data logic can be non trivial.

7.3 Software Development Paradigms

7.3.1 Object Oriented Programming (OOP). Varv's notion of concepts has direct parallels to classes in OOP. Concepts consist of two parts, a schema, and actions. The schema is similar to class properties, and the actions are similar to class methods. Varv's extension methods — `"inject"` and `"join"` — are synonymous with mixins and traits. Because of these parallels, any of the interactive applications built-in Varv could be expressed with OOP. However, there are a few key differences. Object-oriented code is imperative, which leaves less room for the underlying runtime to implement optimizations, and provides a more difficult target for higher-level tooling. Most object-oriented languages do not provide mechanisms for modification or extension of classes without changing the source code. In contrast, Varv concepts are declarative, inherently structured, and support modification via addition. Varv concepts consist of primitives, which enable the high-level yet expressive specification of application and interaction logic. Varv concepts do not provide mechanisms for encapsulation, such as private variables. The lack of encapsulation forces Varv applications to replicate the store design paradigm from Flux applications and aids rapid prototyping. Additionally, Varv concepts support modification and extension via addition, enabling new workflows for the development of interactive applications.

7.3.2 Feature-Oriented Software Development. Incremental [14] or Feature-Oriented Software Development (FOSD) is an area of research that provides mechanisms to incrementally develop software one feature at a time [3]. There are two general approaches to FOSD: compositional and annotative. Compositional approaches enable the development of features in distinct modules that can later be composed to create fully working applications. Most research implements compositional techniques as extensions to existing languages [2, 5, 8] but tools for adding compositional feature development to arbitrary languages exist as well [4, 8]. Annotative approaches enable feature-oriented development using explicit annotations of source code, such as `#ifdef` [35]. In general annotative approaches provide greater flexibility because they allow the modification of source code at the statement level, while compositional approaches provide better organization because code associated with each feature is modular and self contained [3].

Varv concepts implement a compositional approach to and retain similar limitations to past compositional systems. Compositional techniques generally do not provide mechanisms to introduce code fragments where order matters [35]. Within the context of Varv, this means there are certain cases where extending Varv programs requires duplication of existing code. Additionally, programs written using a compositional FOSD paradigm can be difficult to reason about because the final program results from multiple distinct artifacts [35]. Higher-level tools which visualize the final combined program can help [35, 36]. We believe similar tooling could be developed for Varv as well.

7.3.3 Conceptual Design of Software. Software engineers have long realized that they can build more complex and more efficient applications by sharing and reusing software components [52]. Déjà Vu [61] identifies that many web applications are built using combinations of similar components and provides a catalog of self contained and reusable components — called concepts — which can be integrated using a declarative template language to build non trivial applications. Déjà Vu identifies that concept oriented architectures can allow for incremental development of applications by adding one concept at a time and testing functionality. In Déjà Vu the user is able to utilize concepts from a core catalog, and this catalog can be used to implement a wide variety of applications. However, if the user wants to implement their own concepts they need to write a frontend component and a backend server implementation.

Varv's approach, including our choice to name its core building block a "concept," is deeply inspired by Déjà Vu and Jackson's writing on concept design [32, 33]. Varv implements a similar architecture in which concepts are bound to the UI using a template language. However Varv provides a lower level catalog of abstractions, such as actions which can be used for modifying state, and triggers which can be used for listening to state changes or user interactions. Varv focuses on providing declarative mechanisms for users to *compose* lower level abstractions into higher level semantic or domain specific abstractions. Varv also allows users to *extend* existing concepts and *define* new concepts without dropping into JavaScript. By providing mechanisms for extension, Varv allows incremental development one feature at a time.

8 DISCUSSION

8.1 Limitations

Varv is a research prototype and, as such, it does not yet provide all the features necessary for building production-grade interactive software. There are two classes of missing features: those which are straightforward to implement but missing due to time constraints and those which require careful thought and are potential research questions for future work. In the first category are issues such as the lack of support for accessing remote or asynchronous data, the expressive limitations of Varv's templates compared to templates found in popular frameworks such as React or Vue, and the relatively small standard library of actions and events provided by Varv. In the second category are issues such as the lack of access controls, the choice of template languages for alternative substrates, the inability to extend templates via addition, the challenges in authoring incrementally developed applications, and the challenges of supporting polymorphism in Varv. We expand on each of these issues from the second category below.

Information Hiding and Access Controls. Concepts have no notion of private properties, which means any concept can access the properties of any other concept. This lack of information hiding is a conscious design choice because it replicates the Store design pattern — a common approach adopted by frontend libraries (e.g., Redux, Vue, and Svelte) where application state is managed centrally to simplify developing cross-cutting interface elements. In doing so, Varv facilitates rapid prototyping and extension but this limitation makes it challenging to write interactive software that contains private secrets, such as API keys and passwords, or that relies on limiting read or write access to data to specific users, such as chat applications. Varv does offer a limited workaround: users can define local data stores that are not synchronized. These local stores allow users to store things like configurations but are not suitable for secrets since the data is still accessible by other concepts. It remains to be seen if we can augment Varv with a concise, descriptive, and legible syntax for annotating data with identity information and access controls while preserving the rapid prototyping affordances of our current approach.

Definition Files and Templates for Alternative Substrates. Varv is agnostic to the view layer, but the current template files and bindings rely on the existence of a declarative syntax (HTML) for representing the DOM. One of the design goals for Varv is to decouple application logic from interaction modality because we realized early on that it would be valuable to enable the rapid retargeting of interactions from one modality to another. We plan to integrate Varv with substrates outside the DOM environment, such as a WebGL view to support 3D or AR rendering or an IoT substrate that supports declarative interactive logic for intelligent devices such as lights and switches. We believe this is possible but are unaware of declarative template languages for expressing the view or, in the abstract, bindings between concepts and these substrates.

Template Modification Via Addition. Varv's templates support composition via template references, but when users add new features to Varv applications, previous templates and template refs often have to be copied and modified, complicating the development process, duplicating code, and effectively breaking with the open authorial principle [7]. The challenges of supporting template modification via addition may be a limit posed by compositional methods to extension. In compositional approaches to feature-oriented software development, it is considered impossible to introduce statements in the middle of existing methods [35]. If we consider the template definition synonymous with a function definition, this limitation is also applicable to templates. AspectJ provides a unique approach by enabling the extension of method calls within specific methods [38]. However, this multi-level approach to extension can be challenging to reason about and only covers certain cases, such as overriding a nested template in a specific parent template.

Authoring Challenges. While the presented debugging tools are a first step to support authoring in Varv, the nature of accruing changes over time, possibly in many concept definitions and templates, poses new questions regarding how tools can best support authoring incrementally developed applications: If an application is edited by multiple users over longer periods of time and each modification is added through addition, users have to traverse each file, mentally tracking the incremental development of the application's concepts, actions, and triggers, in order to understand the application state. Future tools might support users by making it possible to inspect the current state of an application behavior, i.e. presenting the user the merged concept definition and templates. While this would condense code into a single concept definition and template, such a process might lose information about how and in which order modifications were developed. Providing context and provenance for changes would be important.

Supporting Polymorphism. When Varv injects a source concept into a target concept using extension mechanisms, the target concept inherits the actions and properties of the source concept, but Varv does not form an *is a* relationship between the source and target concept. The target concept cannot populate properties or views which refer to the source concept, even though the target concept exposes the same interface of properties and actions as the source concept. This limitation is due to the event flow model. Allowing multiple concept types to appear in the same set of contexts could create a set of contexts with diverging states, for example, if an action is defined differently in the various concepts. Without polymorphism, it is not easy to create sets of similar objects which each have unique behavior.

Evaluating Usability. We have evaluated the feasibility and expressivity of Varv through demonstration [44]. However, we have not evaluated the usability of the programming model with actual users. A user study of user interface systems such as our work with Varv is challenging [44, 59]. Currently, our tooling for development is proof-of-concept, and an — ideally longitudinal — study of software appropriation over time with Varv would require extensive tool support. Additionally, a user study would be required to understand whether users who are proficient programmers but unfamiliar with declarative programming, can make use of Varv.

We chose an event driven architecture because event architectures are well suited for incremental development [49]. Users can write new actions which run before or after any existing action or UI event in a Varv application, without changing the existing action or UI event. However, over-use of events can inhibit comprehensibility and debuggability of larger programs [49]. Our authoring and debugging tools (see subsection 6.1 and 6.2) let users edit templates and view data associated with concepts, but future work could explore richer visualizations or tracing and debugging of the event graph, potentially easing the developer experience in larger applications. Additionally, a new runtime could explore alternative programming styles such as event-driven functional reactive programming [66], or functional programming, which avoids issues of declaratively managing state.

8.2 Future Work

With Varv we have demonstrated that a declarative approach to specifying interactive applications as data structure is not only possible, but also provides a range of powerful capabilities. Through two very different applications built on top of Varv we have demonstrated that the ceiling for what can be achieved with Varv is high, but we do not clearly know its bounds in terms of expressivity and performance. Thus, an immediate opportunity for future work would be to more systematically evaluate these two aspects.

To better assess Varv's performance, future work could begin by conducting comparative benchmark studies. Following the approaches used to evaluate the performance of frontend JavaScript libraries, these studies could measure both the performance (i.e., time taken) as well as memory consumption of running a suite of operations like rendering, manipulating, and updating thousands of interface elements. Besides empirical methods, future work on performance optimization can also look to practices already adopted by these frontend libraries as well as techniques detailed in the academic literature on dataflow management. For instance, as updating the DOM can be a computationally-intensive operation, React selectively updates DOM nodes by maintaining an in-memory virtual DOM [20]. Similarly, the data stream management community has developed methods for incrementally processing data by flagging data tuples as either new or removed, and only passing these flagged tuples (rather than the full data table) between dataflow nodes [1, 6].

Future work on determining Varv's expressive ceiling can unfold in myriad ways. Our choice of implementing a todo list for our first case study was motivated by TodoMVC [69], which provides a benchmark to compare how various Model-View frameworks implement todo list applications. A next step would then be to target alternate benchmarks such as the seven challenging GUI programming tasks from 7GUIs [39]. To scale this approach, one could turn to large datasets of interactive applications [18] and interaction traces [17] to catalog common classes of interaction techniques, and decompose them into recurring conceptual design patterns — an approach that Déjà Vu has already begun to explore [61]. While promising, these directions adopt primarily qualitatively methods to determine expressivity. An alternate approach might follow McGuffin and Fuhrman [51] to more formally evaluate Varv's expressivity.

An exciting avenue for future work, and a direction inspired by the effect Vega [66] and Vega-Lite [65] have had in data visualization, would explore higher-level systems for authoring Varv applications. In particular, by representing interactive software as a data structure, Varv makes it possible to programmatically reason about the composition of applications. As a result, one can imagine building not only freeform direct manipulation graphical authoring environments (akin to Lyra in the Vega/Vega-Lite ecosystem [64, 73]) but also methods for recommending and autocompleting interaction design (analogous to Data Voyager [72] or Juxxt [68]). For instance, a higher-level system might analyze the schema of concepts currently in use, and execute a lookup in the catalog to identify other concepts that are often used together or that have a complimentary schema. Besides the catalog described in the previous paragraph, such workflows would require the development of additional infrastructure to support a "concept ecosystem," i.e., mechanisms to package and share concept definitions [27].

Such programmatic reasoning about the concepts that underlie interactive software also recalls ideas of instrumental interaction described by Beaudouin-Lafon [9]. Namely, Beaudouin-Lafon envisions a future where interaction techniques — reified [10] as "instruments" — rather than applications are the primary organizational unit of user interfaces. Thus, he imagines that interaction instruments can be reappropriated and used in contexts they were not initially designed for (e.g., using a snap-to-grid feature, typically found in vector graphics packages, but to organize the icons on your desktop). To realize such a vision, however, will require more sophisticated methods to compose and extend concepts than Varv currently supports. Here, one may look to the operators described by Jackson [33] such as action or structure (schema) synchronization, or Project Cambria's [45, 48] approach of bidirectional lenses [23, 29, 30]. Reasoning about and applying the appropriate composition operators automatically would be a critical step on the journey to a cognitively convivial information space [24, 31].

9 CONCLUSION

Modern software development techniques for constructing interactive software typically involve writing imperative code which is packaged and deployed as hermetically-sealed turnkey applications. Writing *extensible* software is an explicit choice and requires careful design choices. In contrast, Varv provides a declarative approach to developing software, yielding an *accretive* development process and applications that are *inherently extensible*. We have outlined the design goals of Varv and have explained how the components of the Varv language — *concepts*, *schema*, *actions* — help fulfill those design goals. We have demonstrated through two case studies the development process enabled by Varv, showing how Varv can be used

to construct a domain specific toolkit for building board games, and how Varv can be used to collaboratively and incrementally develop a shared todo list feature by feature. We provide two examples of higher level tooling built on top of Varv, an inspector for accessing relevant code directly from an application's UI and an alternative Blockly-based editor interface. We hope that Varv inspires future research to enable non-programmers to develop interactive software.

ACKNOWLEDGMENTS

We thank Geoffrey Litt, Daniel Jackson, Philip Tchernavskij, and the anonymous reviewers for their helpful feedback. We also thank Jonas Oxenbøll Petersen for assistance in preparing the video figure. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 740548) and from Carlsbergfondet (grant agreement No CF17-0643).

REFERENCES

[1] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: a new model and architecture for data stream management. *The VLDB Journal—The International Journal on Very Large Data Bases* 12, 2 (2003), 120–139. https://doi.org/10.1007/s00778-003-0095-z

[2] Felipe I. Anfurrutia, Oscar Díaz, and Salvador Trujillo. 2007. On Refining XML Artifacts. In *Web Engineering (Lecture Notes in Computer Science)*, Luciano Baresi, Piero Fraternali, and Geert-Jan Houben (Eds.). Springer, Berlin, Heidelberg, 473–478. https://doi.org/10.1007/978-3-540-73597-7_39

[3] Sven Apel and Christian Kästner. 2009. An Overview of Feature-Oriented Software Development. *The Journal of Object Technology* 8, 5 (2009). https://doi.org/10.5381/jot.2009.8.5.c5

[4] Sven Apel, Christian Kastner, and Christian Lengauer. 2009. FEATUREHOUSE: Language-Independent, Automated Software Composition. In *2009 IEEE 31st International Conference on Software Engineering*. 221–231. https://doi.org/10.1109/ICSE.2009.5070523

[5] Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. 2005. *FeatureC++: Feature-Oriented and Aspect-Oriented Programming in C*. Technical Report.

[6] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. 2004. *STREAM: The Stanford Data Stream Management System*. Technical Report 2004-20. Stanford InfoLab. http://ilpubs.stanford.edu:8090/641/

[7] Antranig Basman, Clayton Lewis, and Colin Clark. 2018. The Open Authorial Principle: Supporting Networks of Authors in Creating Externalisable Designs. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Boston, MA, USA) *(Onward! 2018)*. Association for Computing Machinery, New York, NY, USA, 29–43. https://doi.org/10.1145/3276954.3276963

[8] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. 2004. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering* 30, 6 (2004), 355–371. https://doi.org/10.1109/TSE.2004.23

[9] Michel Beaudouin-Lafon. 2000. Instrumental Interaction: An Interaction Model for Designing Post-WIMP User Interfaces. In *Proceedings of the 18th international conference on Human factors in computing systems*. https://doi.org/10.1145/332040.332473

[10] Michel Beaudouin-Lafon and Wendy E. Mackay. 2000. Reification, Polymorphism and Reuse: Three Principles for Designing Visual Interfaces. In *Proceedings of the working conference on advanced visual interfaces*. https://doi.org/10.1145/345513.345267

[11] Edward Benson, Amy X. Zhang, and David R. Karger. 2014. Spreadsheet Driven Web Applications. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*. ACM, Honolulu Hawaii USA, 97–106. https://doi.org/10.1145/2642918.2647387

[12] Alan F. Blackwell, Carol Britton, Anna Cox, Thomas R. G. Green, Corin Gurr, Gada Kadoda, Maria S. Kutar, Martin Loomes, Chrystopher L. Nehaniv, Marian Petre, et al. 2001. Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. In *International Conference on Cognitive Technology*. Springer, 325–341. https://doi.org/10.1007/3-540-44617-6_31

[13] Marcel Borowski, Janus Bager Kristensen, Rolf Bagge, and Clemens N. Klokmose. 2021. *Codestrates v2: A Development Platform for Webstrates*. Technical Report. Aarhus University. https://pure.au.dk/portal/en/publications/codestrates-v2-a-development-platform-for-webstrates(66e1d4d9-27da-4f6b-85b3-19b0993caf22).html

[14] Mahil Carr. 1997. Prototyping and Software Development Approaches. (1997).

[15] Stefano Ceri, Piero Fraternali, and Aldo Bongio. 2000. Web Modeling Language (WebML): A Modeling Language for Designing Web Sites. *Computer Networks* 33, 1-6 (June 2000), 137–157. https://doi.org/10.1016/S1389-1286(00)00040-2

[16] Kerry Shih-Ping Chang and Brad A. Myers. 2014. Creating Interactive Web Data Applications with Spreadsheets. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. Association for Computing Machinery, New York, NY, USA, 87–96. https://doi.org/10.1145/2642918.2647371

[17] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *Proceedings of the 30th Annual Symposium on User Interface Software and Technology (UIST '17)*. https://doi.org/10.1145/3126594.3126651

[18] Biplab Deka, Zifeng Huang, and Ranjitha Kumar. 2016. ERICA: Interaction Mining Mobile Apps. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (Tokyo, Japan) *(UIST '16)*. ACM, New York, NY, USA, 767–776. https://doi.org/10.1145/2984511.2984581

[19] Donghua Deng, Guigang Zhang, ZhiYuan Gong, Zonglin Guo, and Phillip C-y Sheu. 2008. Semantic Programming of Web-Enabled Database Applications. In *2008 IEEE International Workshop on Semantic Computing and Applications*. 51–60. https://doi.org/10.1109/IWSCA.2008.24

[20] Facebook Inc. 2021. *Virtual DOM and Internals – React*. Retrieved November 25, 2021 from https://reactjs.org/docs/faq-internals.html

[21] Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. 2000. Declarative Specification of Web Sites with Strudel. (2000). https://doi.org/10.1007/s007780050082

[22] Figma Inc. 2021. *Figma*. Retrieved November 25, 2021 from https://www.figma.com

[23] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29, 3 (2007). https://doi.org/10.1145/1232420.1232424

[24] Amy Rae Fox, Philip Guo, Clemens Nylandsted Klokmose, Peter Dalsgaard, Arvind Satyanarayan, Haijun Xia, and James D Hollan. 2020. Towards a Dynamic Multiscale Personal Information Space: Beyond Application and Document Centered Views of Information. In *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*. 136–143. https://doi.org/10.1145/3397537.3397542

[25] Piero Fraternali and Paolo Paolini. 2000. Model-Driven Development of Web Applications: The AutoWeb System. *ACM Transactions on Information Systems* 28, 4 (2000). https://doi.org/10.1145/358108.358110

[26] Google LLC. 2021. *Blockly*. Retrieved November 25, 2021 from https://developers.google.com/blockly

[27] Mona Haraty, Joanna McGrenere, and Andrea Bunt. 2017. Online Customization Sharing Ecosystems: Components, Roles, and Motivations. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing* (Portland, Oregon, USA) *(CSCW '17)*. Association for Computing Machinery, New York, NY, USA, 2359–2371. https://doi.org/10.1145/2998181.2998289

[28] Jeffrey Heer and Michael Bostock. 2010. Declarative Language Design for Interactive Visualization. *IEEE Transactions on Visualization and Computer Graphics* 16, 6 (Nov. 2010), 1149–1156. https://doi.org/10.1109/TVCG.2010.144

[29] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. 2011. Symmetric Lenses. *ACM SIGPLAN Notices* 46, 1 (2011), 371–384. https://doi.org/10.1145/1926385.1926428

[30] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. 2012. Edit Lenses. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 495–508. https://doi.org/10.1145/2103656.2103715

[31] Jim Hollan and Arvind Satyanarayan. 2018. Designing Cognitively Convivial Physics for Dynamic Visual Information Substrates. In *CHI 2018 Workshop on Rethinking Interaction: From Instrumental Interaction to Human-Computer Partnerships*. http://vis.csail.mit.edu/pubs/towards-cognitively-convivial-info-physics

[32] Daniel Jackson. 2015. Towards a Theory of Conceptual Design for Software. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* (Pittsburgh, PA, USA) *(Onward! 2015)*. Association for Computing Machinery, New York, NY, USA, 282–296. https://doi.org/10.1145/2814228.2814248

[33] Daniel Jackson. 2021. *The Essence of Software: Why Concepts Matter for Great Design*. Princeton University Press.

[34] Szymon Kaliski, Adam Wiggins, and James Lindenbaum. 2019. *End-User Programming*. Retrieved November 25, 2021 from https://www.inkandswitch.com/end-user-programming.html

[35] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2008. Granularity in Software Product Lines. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. 311–320. https://doi.org/10.1145/1368088.1368131

[36] Christian Kästner, Thomas Thum, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. 2009. FeatureIDE: A Tool Framework for Feature-Oriented Software Development. In *2009 IEEE 31st International Conference on Software Engineering*. 611–614. https://doi.org/10.1109/ICSE.2009.5070568

[37] Alan Kay and Adele Goldberg. 1977. Personal Dynamic Media. *Computer* 10, 3 (March 1977), 31–41. https://doi.org/10.1109/C-M.1977.217672

[38] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *ECOOP 2001 — Object-Oriented Programming (Lecture Notes in Computer Science)*, Jørgen Lindskov Knudsen (Ed.). Springer, Berlin, Heidelberg, 327–354. https://doi.org/10.1007/3-540-45337-7_18

[39] Eugen Kiss. 2021. *7GUIs*. Retrieved November 25, 2021 from https://eugenkiss.github.io/7guis/

[40] Martin Kleppmann. 2017. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems* (1st edition ed.). O'Reilly Media, Boston.

[41] Clemens N. Klokmose, James R. Eagan, Siemen Baader, Wendy E. Mackay, and Michel Beaudouin-Lafon. 2015. Webstrates: Shareable Dynamic Media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software and Technology*. https://doi.org/10.1145/2807442.2807446

[42] Amy J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. 2011. The State of the Art in End-User Software Engineering. *Comput. Surveys* 43, 3 (April 2011), 1–44. https://doi.org/10.1145/1922649.1922658

[43] Butler Lampson. 2020. Hints and Principles for Computer System Design. (2020).

[44] David Ledo, Steven Houben, Jo Vermeulen, Nicolai Marquardt, Lora Oehlberg, and Saul Greenberg. 2018. *Evaluation Strategies for HCI Toolkit Research*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3173574.3173610

[45] Geoffrey Litt, Peter van Hardenberg, and Orion Henry. 2021. Cambria: Schema Evolution in Distributed Systems with Edit Lenses. In *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*. https://doi.org/10.1145/3447865.3457963

[46] Geoffrey Litt and Daniel Jackson. 2020. Wildcard: Spreadsheet-Driven Customization of Web Applications. In *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming* (Porto, Portugal) *(<programming> '20)*. Association for Computing Machinery, New York, NY, USA, 126–135. https://doi.org/10.1145/3397537.3397541

[47] Geoffrey Litt, Daniel Jackson, Tyler Millis, and Jessica Quaye. 2020. End-User Software Customization by Direct Manipulation of Tabular Data. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Virtual, USA) *(Onward! 2020)*. Association for Computing Machinery, New York, NY, USA, 18–33. https://doi.org/10.1145/3426428.3426914

[48] Geoffrey Litt, Peter van Hardenberg, and Orion Henry. 2020. *Project Cambria: Translate your data with lenses*. Retrieved November 25, 2021 from https://www.inkandswitch.com/cambria.html

[49] Cristina Videira Lopes. 2020. *Exercises in Programming Style* (second edition ed.). CRC Press, Boca Raton.

[50] Matt McCutchen, Shachar Itzhaky, and Daniel Jackson. 2016. Object Spreadsheets: A New Computational Model for End-User Development of Data-Centric Web Applications. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, Amsterdam Netherlands, 112–127. https://doi.org/10.1145/2986012.2986018

[51] Michael J. McGuffin and Christopher P. Fuhrman. 2020. Categories and Completeness of Visual Programming and Direct Manipulation. In *Proceedings of the International Conference on Advanced Visual Interfaces* (Salerno, Italy) *(AVI '20)*. Association for Computing Machinery, New York, NY, USA, Article 7. https://doi.org/10.1145/3399715.3399821

[52] Douglas M. Mcllroy. 1968. Mass-Produced Software Components. *Proceedings of the 1st International Conference on Software Engineering* (1968), 88–98.

[53] MDN Contributors. 2021. *Array.prototype.map() - JavaScript | MDN*. Retrieved November 25, 2021 from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

[54] MDN Contributors. 2021. *Template Literals (Template Strings) - JavaScript | MDN*. Retrieved November 25, 2021 from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

[55] Giansalvatore Mecca, Paolo Atzeni, Alessandro Masci, Giuseppe Sindoni, and Paolo Merialdo. 1998. The Araneus Web-Based Management System. *SIGMOD Rec.* 27, 2 (jun 1998), 544–546. https://doi.org/10.1145/276305.276375

[56] Midas Nouwens and Clemens Nylandsted Klokmose. 2021. A Survey of Digital Working Conditions of Danish Knowledge Workers. In *Proceedings of 19th European Conference on Computer-Supported Cooperative Work*. European Society for Socially Embedded Technologies (EUSSET). https://doi.org/10.18420/ecscw2021_n24

[57] Observable, Inc. 2021. *Observable*. Retrieved November 25, 2021 from https://observablehq.com

[58] Yoshiki Ohshima, Aran Lunzer, Bert Freudenberg, and Ted Kaehler. 2013. KScript and KSWorld: A Time-Aware and Mostly Declarative Language and Interactive GUI Framework. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. 117–134. https://doi.org/10.1145/2509578.2509590

[59] 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. https://doi.org/10.1145/1294211.1294256

[60] OpenJS Foundation. 2021. *Electron*. Retrieved November 25, 2021 from https://www.electronjs.org

[61] Santiago Perez De Rosso, Daniel Jackson, Maryam Archie, Czarina Lao, and Barry A. McNamara III. 2019. Declarative Assembly of Web Applications from Predefined Concepts. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Athens, Greece) *(Onward! 2019)*. Association for Computing Machinery, New York, NY, USA, 79–93. https://doi.org/10.1145/3359591.3359728

[62] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. 2016. Foundations of JSON Schema. In *Proceedings of the 25th International Conference on World Wide Web* (Montréal, Québec, Canada) *(WWW '16)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 263–273. https://doi.org/10.1145/2872427.2883029

[63] Ian Piumarta and Kimberly Rose. 2010. *Points of View: A Tribute to Alan Kay*. Viewpoints Research Institute, Glendale, Calif.

[64] Arvind Satyanarayan and Jeffrey Heer. 2014. Lyra: An Interactive Visualization Design Environment. *Computer Graphics Forum* 33, 3 (2014), 351–360. https://doi.org/10.1111/cgf.12391

[65] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (2017), 341–350. https://doi.org/10.1109/TVCG.2016.2599030

[66] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. 2016. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE Transactions on Visualization and Computer Graphics* 22, 1 (2016), 659–668. https://doi.org/10.1109/TVCG.2015.2467091

[67] Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. 2014. Declarative Interaction Design for Data Visualization. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology* (Honolulu, Hawaii, USA) *(UIST '14)*. Association for Computing Machinery, New York, NY, USA, 669–678. https://doi.org/10.1145/2642918.2647360

[68] Kesler Tanner. 2019. *Visual Design Tools in Support of Novice Creativity*. Stanford University.

[69] TodoMVC. 2021. *TodoMVC*. Retrieved November 25, 2021 from http://todomvc.com

[70] Jacob VanderPlas, Brian Granger, Jeffrey Heer, Dominik Moritz, Kanit Wongsuphasawat, Arvind Satyanarayan, Eitan Lees, Ilia Timofeev, Ben Welsh, and Scott Sievert. 2018. Altair: Interactive Statistical Visualizations for Python. *Journal of Open Source Software* 3, 32 (Dec. 2018). https://doi.org/10.21105/joss.01057

[71] Lea Verou, Amy X. Zhang, and David R. Karger. 2016. Mavo: Creating Interactive Data-Driven Web Applications by Authoring HTML. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (Tokyo, Japan) *(UIST '16)*. Association for Computing Machinery, New York, NY, USA, 483–496. https://doi.org/10.1145/2984511.2984551

[72] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. 2016. Voyager: Exploratory Analysis via Faceted Browsing of Visualization Recommendations. *IEEE Transactions on Visualization and Computer Graphics* 22, 1 (Jan. 2016), 649–658. https://doi.org/10.1109/TVCG.2015.2467191

[73] Jonathan Zong, Dhiraj Barnwal, Rupayan Neogy, and Arvind Satyanarayan. 2020. Lyra 2: Designing Interactive Visualizations by Demonstration. *IEEE Transactions on Visualization and Computer Graphics* 27, 2 (2020), 304–314. https://doi.org/10.1109/TVCG.2020.3030367

A VARV LANGUAGE EXAMPLE

A.1 Abstract Concept Definition

```
{
  "concepts": {
    "todo": {
      "schema": { "label": "string" }
    },
    "todoList": {
      "schema": {
          "todos": { "array": "todo" },
          "todosCount": { "number": {
              "derive": {
                "properties": [ "todos" ],
                "transform": [
                    { "length": "todos" }
                ]
              }
          }}
      },
      "actions": {
        "addNewTodo": [
          { "new": {
              "concept": "todo",
              "with": {
                  "label": "@newTodoLabel"
              },
              "as": "newTodo"
          }},
          { "append": {
              "property": "todoList.todos",
              "item": "$newTodo"
          }}
        ]
      }
    },
    "todoInput": {
      "schema": {
        "text": "string"
      },
      "actions": {
        "activateInput": [
          { "get": {
              "property": "todoInput.text"
          }},
          { "set": { "text": "" }},
          { "addNewTodo": {
              "newTodoLabel": "$get"
          }}
        ]
      },
      "mappings": {
        "text": [ "memory" ]
      }
    }
  }
}
```

Listing 1: Example of an abstract concept definition of a todo list app consisting of three concepts.

A.2 Concrete Concept Definition

```
{
  "concepts": {
    "todo": {
      "actions": {
        "deleteOnClick": {
          "when": { "click": {
              "view": "deleteButton"
          }},
          "then": "remove"
        }
      }
    },
    "todoInput": {
      "actions": {
        "activateInput": {
          "when": [
            { "key": {
              "key": "Enter",
              "focus": "todoInput"
            }},
            { "click": {
                "view": "addTodoButton"
            }}
          ]
        }
      }
    }
  }
}
```

Listing 2: Example of an concrete concept definition of a todo list app.

A.3 Template

```
<dom-view-template>
  <template name="todo">
    <div>
      <span class="text">{text}</span>
      <span view="deleteButton">Delete</span>
    </div>
  </template>
  <h2>Todo List</h2>
  <div concept="todoInput">
    <h3>Add New Todos</h3>
    <input value="{text}" />
    <button view="addTodoButton">Add Todo</button>
  </div>
  <div concept="todoList">
    <h3>Todos ({todosCount})</h3>
    <div class="list">
      <div property="todos">
        <template-ref template-name="todo">
        </template-ref>
      </div>
    </div>
  </div>
</dom-view-template>
```

Listing 3: Example of a template of a todo list app.

A.4 Custom Action

```
/**
 * Usage: define a function "foo" in global scope.
 * { "customJS": { "func": "foo" }}
 */
class CustomJSAction extends Action {
  constructor(name, options) {
    if (typeof options === "string") {
      options = { func: options };
    }
    super(name, options);
  }

  async apply(contexts, actionArguments) {
    if (this.options.func == null) {
      throw new Error("'func' must be set");
    }

    return this.forEachContext(
      contexts,
      actionArguments,
      async (context, options) => {
        let func = options.func;

        if (window[func] == null) {
          throw new Error(`'${func}' not defined`);
        }

        if (typeof window[func] !== "function") {
          throw new Error(`'${func}' is not a function`);
        }

        return window[func](context, options);
      }
    );
  }
}
Action.registerPrimitiveAction("customJS",
                               CustomJSAction);
window.CustomJSAction = CustomJSAction;
```

Listing 4: Example action which lets users run arbitrary global functions as actions.

B LIST OF PRIMITIVE ACTIONS

B.1 Concept Actions

Action Name	Description
`"count"`	Returns the count of instances of a given concept type. Filtering like in `"where"` is possible.
`"exists"`	Returns a boolean variable of whether there exist instances of a given concept type. Filtering like in `"where"` is possible.
`"get"`	Returns the value of a property of either the current target or from another concept instance.
`"new"`	Creates a new instance of a given concept with the given properties. Has an option to not select the newly created instance.
`"remove"`	Removes the current target concept instance or instances stored in a variable.
`"set"`	Sets the value of a property or variable.

B.2 Control Flow Actions

Action Name	Description
`"eval"`	Returns the boolean value of a filtering expression.
`"exit"`	Terminates the action chain.
`"limit"`	Limits the number of context to a given count starting from the first or last.
`"run"`	Runs an action with a copy of the current event and then continues with the action chain independent of the outcome of the other action.
`"select"`	Selects all instances of a given concept type. Filtering like in `"where"` is possible.
`"storeSelection"`	Stores the current selected targets in a variable in the event.
`"switch"`	Tests conditions for several branches and executes the action chain of the branch that matches. A default branch can be set and it contains an option to continue after a successful branch.
`"wait"`	Stops and waits with continuing the action chain for a given duration.
`"where"`	Filters the current selection according to a given property or variable condition. Conditions can be combined using "and", "or", and "not."

B.3 Boolean Actions

Action Name	Description
`"toggle"`	Inverts a boolean property or variable.

B.4 String Actions

Action Name	Description
`"concat"`	Concatenates an array of strings or variables to a new string.
`"enums"`	Returns an array of all possible enums of a string property.
`"length"`	Returns the length of a string.
`"textTransform"`	Transforms a string to uppercase, lowercase, or capitalization.

B.5 Number Actions

Action Name	Description
`"calculate"`	Calculates a given mathematical expression.
`"decrement"`	Decrements a number property or variable by a given value.
`"increment"`	Increments a number property or variable by a given value.
`"random"`	Returns a random number within a given range.

B.6 Array Actions

Action Name	Description
`"append"`	Appends an item to an array property or variable.
`"items"`	Returns the items of an array. Allows for filtering items.
`"length"`	Returns the length of an array.
`"prepend"`	Prepends an item to an array.
`"removeFirst"`	Removes the first item from an array.
`"removeItem"`	Removes the item on the given index from an array.
`"removeLast"`	Removes the last item from an array.

C LIST OF PRIMITIVE TRIGGERS

C.1 Reactive Triggers

Trigger Name	Description
`"action"`	Triggers when another given action is executed. Contains an option to specify whether it should trigger before or after the other action was executed.
`"interval"`	Triggers in intervals after a given time.
`"stateChanged"`	Triggers when a given concept or property changes.

C.2 View Triggers

Trigger Name	Description
`"click"`	Triggers when a given concept, property, or view is clicked.
`"key"`	Triggers if a given key is pressed and optionally a concept, property, or view is in focus.
`"mouseDown"`	Triggers when a given concept, property, or view receives a mouseDown event.
`"mouseMove"`	Triggers when a given concept, property, or view receives a mouseMove event.
`"mouseUp"`	Triggers when a given concept, property, or view receives a mouseUp event.

D LIST OF DATA STORES

Data Store Name	Description
`"dom"`	Stores data in the DOM of a website or webstrate. Listens to changes to these elements.
`"memory"`	Stores data in memory that gets emptied after a page refresh.
`"localStorage"`	Stores data in the localStorage.

E LIST OF EXTENSIONS

Extension Name	Description
`"inject"`	Adds the schema and actions of one or multiple source concepts into a target concept. Properties or actions with the same name are overwritten by injected concepts in the order they are specified.
`"join"`	Combines one or multiple source concepts into a new concept. Properties and actions with the same name are handled like in the `"inject"` extension.
`"omit"`	Removes the given properties and actions from a target concept.
`"pick"`	Takes the given properties and concepts of a source concept and creates a new concept based on those.

F LIST OF DOM VIEW TEMPLATE TAGS

F.1 Template Tags

Tag Name	Description
`<dom-view-template>`	Indicates the start and end of a template in the DOM view.
`<template name="some-name">`	Allows to create named templates that can be reused using template references.
`<template-ref name="ref-name">`	Allows to reference a named template and insert it.

F.2 Template Attributes

Attribute Name	Description
`concept`	Indicates that the given element should be rendered for each instance of the given concept.
`property`	Indicates that the given element refers either to a property with type concept or to an array concept.
`view`	Indicates that the given element is a view that can be referred to in concept definition files.
`value`	Indicates that the given value of an text input, select or checkbox should be synchronized with the given property.