The Ultimate GUI Framework: Are We There Yet?

Knut Anders Stokke[1]
University of Bergen, Norway

Mikhail Barash
University of Bergen, Norway

Jaakko Järvi
University of Turku, Finland

Abstract

The programming community seems to be forever searching for the ultimate user interface programming approach and the accompanying framework. We describe the landscape of recent efforts in this quest through describing commonalities and differences of modern JavaScript frameworks with respect to their approaches to GUI specifications. We situate both Eelco Visser's work on WebDSL and our own work on GUI programming in this landscape, and point out areas where more research is needed, including modeling multi-way dataflows and dynamic structures in GUIs.

2012 ACM Subject Classification Software and its engineering → Graphical user interface languages; Software and its engineering → Software libraries and repositories

Keywords and phrases Web frameworks, GUI specification, WebDSL, HotDrink, WarmDrink

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.25

Acknowledgements The authors are thankful to the anonymous reviewers for their valuable comments on the paper. We also wish to express our gratitude for and admiration of Eelco Visser's life's work on programming research, and of him as a person.

1 Introduction

Since the introduction of the Model-View-Controller pattern [32], the programming community has been chasing after better ways to implement graphical user interfaces (GUIs) – to arrange how events are delivered and handled and how source code related to GUIs is organized. We have seen many new patterns [29, 18, 14] and a large number of *frameworks* (we will list many in this paper) over the years. New GUI patterns and frameworks have at times come with some hype, and have been met with excitement that the new way to program GUIs is not merely better, but rather the best, or even the ultimate way of programming GUIs. Such excitement has gradually faded with the introduction of new ultimate GUI frameworks.

How far have we come since 1979, when MVC was proposed? Are we close to the ultimate framework, the final truth in GUI programming? Or are there still features and aspects of GUI programming that can be improved, even significantly? We claim the answer to the last question is positive, and point to at least two such areas.

First, none of the widely used modern GUI frameworks support complex *data dependencies* between variables, which occur in *data-rich* user interfaces. An example of this is a dialog for image resizing: editable variables include the absolute width and height in pixels, the relative width and height in percentages, and the ratio. The program must maintain several relations between all these variables whenever a user updates one of them. Many modern GUI frameworks support functional, but not relational (*multi-way*), dependencies between

[1] Corresponding author.

variables. When application programmers decompose relational dependencies into functional and manage themselves which dependencies should be in effect, GUI logic tends to become scattered throughout event handling functions [22].

Second, GUIs present all kinds of *structures* (such as lists, grids, and trees) to users. There are standard operations for manipulating such structures that users would like to take advantage of, but often GUIs do not provide such tools to the user. Our oft-used example is *ApplyTexas*[2], a website handling admission applications to higher education institutions in the State of Texas. This GUI asks users to provide a list of extracurricular activities, each consisting of more than 20 input fields, in the order of importance, but offers no operations to reorder activities [16]. Modern GUI frameworks lack direct support for generic reusable structure manipulation operations; providing such operations is the responsibility of the application programmer.

This paper surveys a large number of recent popular GUI frameworks and forms a landscape of the different approaches and essential features that can be identified in these frameworks. Further, it argues that the (lack of) quality in today's GUIs is an indication that the ultimate GUI framework or GUI programming paradigm has not yet been discovered, and then discusses challenges, focusing on the two we mentioned above, of GUI programming that today's popular frameworks do not provide answers for. We also briefly describe our own efforts in addressing those challenges and relate our work to Eelco Visser's WebDSL [20], which is a collection of domain-specific languages and tools for developing web applications.

2 The Landscape of JavaScript GUI Frameworks

Frameworks provide a standardized way of developing software through *inversion of control* [8]: the program's control flow is dictated by the framework and not the programmer, who only provides relevant code to get behaviour specific to the application at hand. Frameworks separate the data layer (the "*model*") from the presentation layer (the "*view*") and define how the two interact with each other; this is known as *separation of concerns* [19]. Client-side web frameworks exhibit different approaches to this, and these approaches dictate how user interfaces are *specified*. Usually the programmer specifies the view declaratively, with references to the model's variables. When the variables change, these views are dynamically updated. However, the approaches to update the views and track variable changes differ between frameworks.

To capture a *landscape* of modern GUI frameworks, we gather properties and features where these frameworks manifest different design choices, and characterize each framework through these properties. The cross-tabulation of the properties and frameworks is given in Table 1. The set of frameworks we included is not an exhaustive collection of all JavaScript frameworks; our goal was to include the influential and popular frameworks that have appeared since the early 2010's, but of course the selection is subjective[3].

The common task of all these frameworks is to keep an application's view in sync with its model. The *model* is a collection of data that users manipulate through the user interface, and, for web applications, the *view* is the website's Document Object Model (DOM). Frameworks that update the DOM automatically on model changes are called *reactive*.

[2] `https://www.applytexas.org/`
[3] Certainly, there are other relevant GUI frameworks worth analysing in our setting that are based on different languages than JavaScript, most prominently perhaps SwiftUI [39] and Flutter [10].

Table 1 A comparison of widely used JavaScript frameworks, the constraint system-powered library HotDrink, WebDSL, and the properties that these support. A fully supported property is denoted by ●, a partially supported property is denoted by ◒, and non-supported properties are denoted by ○. The property columns have the following meaning: *declarative view specification* means that views are specified declaratively; *re-rendering mechanism* specifies the approach used to re-render the view; *imperative rendering* means that DOM elements are updated imperatively; *virtual DOM* means that a virtual DOM is used to identify which elements need to change; *compiled rendering* means that a view specification is used to generate code for updating DOM elements; *view replacement* means that on re-render, the entire component view is replaced by a new one, instead of being updated; *two-way binding* means that such bindings are supported; *stateful components* means that components have self-contained state; *component hierarchy* means that components can have sub-components and pass data to them; *multi-way dataflow* means that the reactive system supports multi-way dataflow; *consistent concurrency* means that the reactive system handles asynchronous computations consistently [11]; *semantic model* means that high-level abstract GUI structures are defined; *tracking mechanism* specifies the approach used to track model updates; *setter function* means that the model is updated through a setter function; *messages (MVU)* means that GUI events produce messages to a global update function; *observables* means that model attributes are wrapped in observables (tracked objects) that notify subscribers on value change; *checked after events* means that on every event handling, the model is checked for changes; *compiled tracking* means that model updates are compiled to instructions that notify the system after the update.

Property / Framework	Declarative view spec. [23]	Re-rendering mechanism				Two-way bindings [43]	Stateful components [35]	Component hierarchy [42]	Multi-way dataflow [21]	Consistent concurrency [11]	Semantic model [13]	Tracking mechanism				
		Imperative rendering	Virtual DOM [44]	Compiled rendering [28]	View replacement							Setter-function [35]	Messages (MVU) [6]	Observables [17, 26]	Checked after events	Compiled tracking [1]
Angular [2]	●	○	○	●	○	●	●	●	○	○	○	○	○	○	●	○
Backbone.js [3]	◒	●	○	○	●	○	●	○	○	○	○	●	○	○	○	○
Elm [9]	●	○	●	○	○	○	○	●	○	○	○	○	●	○	○	○
Ember.js [7]	●	○	○	○	●	●	●	●	○	○	○	○	○	●	○	○
Knockout.js [25]	●	●	○	○	○	●	●	●	○	○	○	○	○	●	○	○
React [30]	●	○	●	○	○	○	●	●	○	○	○	●	○	○	○	○
SolidJS [34]	●	○	○	●	○	○	●	●	○	○	○	●	○	●	○	○
Svelte [38]	●	○	○	●	○	●	●	●	○	○	○	○	○	○	○	●
Vue.js [45]	●	○	●	○	○	●	●	●	○	○	○	○	○	●	○	○
HotDrink [15, 36, 37]	◒	●	○	○	○	●	●	●	●	●	●	○	○	●	○	○
WebDSL [20]	●	n/a	n/a	n/a	n/a	●	○	●	○	○	●	n/a	n/a	n/a	n/a	n/a

Backbone.js [3] is one of the oldest among the compared frameworks. As most of the frameworks discussed in this section, it enables programmers to build user interfaces from *components* that are rendered individually. For each component, one needs to implement a rendering function that renders the component using attributes from the model; on model changes, the component is re-rendered and the old DOM is replaced with the new one. Such a rendering mechanism is different from the approach of Angular [2], where component specifications are *compiled*. During the compilation process, the framework derives how each

DOM element depends on the model attributes, and generates functions to update only those elements that need to change. After an event (e.g., a mouse click, a keypress, etc.) has been handled – possibly resulting in updates to the model – Angular compares the current model with the old one and updates the corresponding DOM elements.

Most JavaScript frameworks support one-way bindings in components' view specifications: these bindings update DOM-elements on model changes. Angular, however, enables *two-way bindings* [43]. Such a binding between a model attribute and a user-editable value (e.g., the text in an input field, or a slider's position) means that the framework keeps the two in sync. In frameworks without support for two-way bindings, event handlers are needed to update the model when a user modifies values in the view. In addition to Angular, two-way bindings are supported in the Ember.js [7] and Knockout [25] frameworks. The former uses the design philosophy of *convention over configuration* [46] by providing built-in code for common web application functionality.[4] The latter uses the model-view-viewmodel architecture [9] in order to separate domain data (the *model*), the *view*, and data presented and edited in the view (the *viewmodel*). Knockout treats the (view)model differently than Angular and Backbone.js: instead of using plain JavaScript objects, it wraps data in *observable* objects, which notify their *observers*[5] whenever their value gets updated. That is, when a DOM-element is bound to an observable (view)model attribute, an observer updates the DOM-element on change notification. This distinguishes Knockout from other frameworks that have to compute the changes to the model – or, for that matter, to the view – after every (user) event has been handled.

An entirely different approach to GUI specifications is used in Elm [9], which is a *purely functional* Haskell-like domain-specific language (rather than a framework). Elm introduces the model-view-update (MVU) architecture [14], where the model is an immutable data structure and the view is a function that maps this structure to a DOM. An Elm application has an update function which is called every time an event is fired. This function takes a model and an event as arguments, and it produces a new updated model, which is further used to re-render the view. This architecture can be utilized by other frameworks, such as React [30]. Elm and React, together with another JavaScript framework Vue.js [45], use a rendering method that constructs a new *virtual* DOM each time an update to the model has been made. This new DOM is then *reconciled* with the current DOM [27], that is, the new and the old views are compared in order to find the least number of edits to the view needed to reflect the updated model. After the comparison is made, the necessary modifications are introduced into the actual DOM. This is different from other frameworks (such as Backbone.js), which replace the *entire* current DOM with an updated one. Another distinct feature of Elm is the way it handles *modularity*: rather than being composed from components, GUIs are only specified with pure functions. The entire application model is persisted in one immutable data structure, and all events are handled in a single update function. This is in contrast to React and Vue.js, where state is maintained at the component level and not at the application level[6]. Frameworks Angular, Svelte [38], and SolidJS [34] similarly handle state changes at the component level, but compile component specifications to lower-level JavaScript, and perform static analysis of GUI specifications in order to find dependencies between DOM elements and model attributes. With Svelte, for instance, when a component state attribute is referenced in a component's view specification, the

[4] This is similar to the approach of the back-end framework Ruby on Rails [40].
[5] An observer is a function that takes one argument, the updated value, and performs a side-effect with it.
[6] With the *Redux* [31] extension, state can be handled at the application level also in React.

compiled component code has references to the component's DOM elements and has an update-function; the function takes (updated) component states as argument and updates the state attribute into the corresponding DOM element [28].

3 Multi-Way Dataflow and Structural Modifications

One of the early use cases for JavaScript was to enable *dynamic* client-side changes on websites [48]. While web-servers could construct HTML documents dynamically on request, JavaScript could run on the client-side, react to user interaction, and modify content without refreshing the page, thus enabling a more interactive user experience on the web. JavaScript scripts can show validation messages to users while they type into a text field, update widgets as a user moves a slider, give up-to-date information about the weather forecast as the user types in a city name, and so on. All changeable GUI values can be considered as *variables*, with dependencies between them that should be maintained whenever there is a change to any of the variables. Reacting to such changes is traditionally done in *event handlers* that are triggered by user events; maintaining dependencies directly in event handlers leaves a large coordination effort to the application programmer: *all* event handlers become responsible for *all* GUI variables, and introducing new GUI variables or relations may necessitate modifications to several event handlers. Event handlers with asynchronous tasks require even more involved modifications: before updating a variable, such a handler must ensure that the variable is not currently used as input to an asynchronous computation, in order to keep the GUI in a consistent state [12].

Reactive programming [5, 47] simplifies maintaining dependencies between GUI variables. In this programming model, an underlying system is made aware of all variable dependencies in the GUI, so that every variable change can be propagated to all variables that depend on the changed variable. Most frameworks discussed in Section 2 have some support for reactive programming [3, 25, 2, 7, 9, 30, 45, 38, 34], but they only support one fixed dataflow. While this does not constitute a problem in GUIs where it is known in advance which variables are fixed and which need to be computed, data-rich user interfaces often require support for multiple ways to update variables.

As a familiar example, consider a form for booking a hotel room. The form has fields for the arrival and departure dates, and the number of nights. Some users may want to fill in the first two fields and have the last one computed, while others may prefer to provide the arrival date and number of nights, and have the departure date computed for them. Many hotel-booking applications support only one fixed dataflow, and show some of the variables as non-editable GUI elements (if showing them at all). A possible reason for the prevalence of such feature-limited user interfaces is the accidental complexity [4] of implementing and orchestrating multiple dataflows by hand.

An alternative to maintaining dependencies manually is to define them as constraints in a *constraint system*. HotDrink [15] is an example of a *multi-way dataflow constraint system*-powered library that enables specifying relations, multi-way dependencies, between variables. Each constraint in such a specification has satisfaction methods that enforce the constraint when executed. Whenever a variable is updated, the library solves the constraint system [33] by choosing and executing one method from each constraint, in order to satisfy all the relations between the variables. In addition, the history of the last edited values is maintained, and is used to find dataflows that are least "surprising" to the user [12].

Listing 1 specifies a constraint component for the hotel booking example using the HotDrink DSL. The first constraint represents the relation that `nights` is the number of nights between `arrival` and `departure`, and has three satisfaction methods to enforce

Listing 1 A HotDrink specification of multi-way dependencies in a GUI.

```
component HotelBooking {
  var arrival, departure, nights, showErrorMessage;
  constraint {
    (arrival, departure -> nights) =>
      Math.ceil((date2.getTime()-date1.getTime())/(1000*3600*24));
    (arrival, nights -> departure) => ...;
    (departure, nights -> arrival) => ...;
  }
  constraint {
    (nights -> showErrorMessage) => nights <= 0;
  } }
```

this constraint, thus enabling all three variables to be set by a user. The fourth variable, `showErrorMessage`, is a boolean variable that is true if a stay would have zero or a negative number of nights, as specified in the last constraint (this variable can not be set by the user). A user interface with input fields for specifying hotel bookings can be connected with this component (with two-way bindings between input fields and variables) to ensure that the relation between the fields is maintained. Additionally, the view can subscribe to the observable variable `showErrorMessage` and use it to toggle the display of an error message on invalid booking dates.

The core Hotdrink library allows the programmer to specify constraint system *components*, collections of variables and constraints. Further, it has a (low-level) API for linking components via sharing of variables. Implementing structural operations using this API is, however, error-prone, as operations must handle many different scenarios. For instance, in a list where each component is connected to the succeeding component, removing the first component involves different constraint system updates than removing the last component, or removing a component in between two others. And of course, if the component specification itself changes, structural operations might have to be modified.

An extended version of the library, called WarmDrink [36, 37], builds on these low-level operations and enables composing components into *structures*, such as lists and trees, with data flowing between the components. With WarmDrink the programmer specifies concisely how constraint components are related within a structure. Based on this specification WarmDrink generates a high-level API for modifying the GUI structure, e.g., an API for adding, removing and swapping connected components in lists. Equipped with such an API, the application programmer can easily provide a rich set of structure manipulation tools to the user.

4 Relating to WebDSL

The idea of generating full GUI implementations from concise high-level specifications of structure – similarly to what WarmDrink does – is notably implemented in *WebDSL* [20], which is a set of domain-specific languages for defining web applications, together with a static analysis tool that performs cross-language validation. These languages focus on defining entities (i.e., the *model*), pages (i.e., the *view*), access control, and various actions (e.g., persisting data when users submit a form). The WebDSL's static analyser checks, for example, the existence of properties displayed in a GUI, or the existence of pages defined in access control specifications.

Though WarmDrink and WebDSL solve different problems, their goal is to reduce the accidental complexity of web development, especially of form-based web applications. Both frameworks maintain relations between model attributes: while HotDrink enables specifying constraints on the variables and WarmDrink enables specifying relations between GUI components, WebDSL enables specification of aggregation relationships between entities. Both HotDrink and WebDSL support two-way bindings (as do Knockout.js, Angular and Svelte). WebDSL takes such bindings one step further than other frameworks: pages can call method `persist` on an entity that a user has edited, and this updated entity will be saved in the application's database. This conceptually binds the values of DOM elements to values persisted on a server.

As with most other frameworks discussed in this paper, when using WebDSL the view is specified with declarative HTML-templates, rather than by constructing DOM-elements with JavaScript code. A major departure from other frameworks is, however, that WebDSL renders web pages on the server, while the frameworks perform rendering in the browser. The latter approach enables web pages to update dynamically when the user navigates between them, but it also involves downloading the framework source code and starting the framework runtime in the browser, which has significant overhead compared to viewing the content of a plain HTML file.[7] Because of the overhead of client-side rendering, a recent trend is to run JavaScript frameworks on the server. This approach, known as *server-side rendering* [41], combines the expressiveness and modularity of JavaScript frameworks with the performance of server-side rendered websites, and also shifts JavaScript frameworks closer to WebDSL.

While data-rich web applications can be rendered on a server, users still do interact with input forms and widgets that affect each other's behaviour, and this behaviour should be updated *dynamically*. For instance, when filling out a form, users benefit from getting validation messages as they type, not after they have submitted the form. If the form additionally involves a *structure* of repeated fields, the GUI should provide operations for modyifying that structure. Observations today [37, 36] highlight that such crucial operations are often missing even in widely used web applications.

Many approaches and frameworks, WebDSL and our HotDrink and WarmDrink amongst them, have over the years made significant contributions to increase our understanding about GUI programming. We hope, however, that we have made it clear that there is (still) room and a need for more research on GUI programming. We are certainly continuing our investigations on multi-way dataflow constraint systems for GUIs, to move us closer to the, perhaps elusive, goal of "the ultimate GUI framework". Finally, while the WebDSL and JavaScript frameworks with server-side rendering enable embedding of JavaScript code into client-side HTML, making dynamic changes on a page, such code is typically self-contained and not statically checked against the rest of the source code. Perhaps addressing these issues could be a WebDSL-inspired avenue of inquiry towards that same goal.

References

1 Advanced Svelte: Reactivity, lifecycle, accessibility. Accessed 26.10.2022. URL: `https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Svelte_reactivity_lifecycle_accessibility`.

[7] A blog post [24], surveying thousands of websites that use different JavaScript frameworks, reports that a median Angular website for mobile devices involves downloading 1.1 MB of JavaScript code and takes 4.1 seconds of scripting-related CPU time. For the 90th percentile, the numbers are 2.9 MB and 13.3 seconds, respectively.

2 Angular. Accessed 26.10.2022. URL: https://angular.io/.

3 Backbone.js. Accessed 26.10.2022. URL: https://backbonejs.org/.

4 Frederick P. Brooks, Jr. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987. doi:10.1109/MC.1987.1663532.

5 Conal Elliott and Paul Hudak. Functional reactive animation. *SIGPLAN Not.*, 32(8):263–273, August 1997. doi:10.1145/258949.258973.

6 The Elm architecture. Accessed 26.10.2022. URL: https://guide.elm-lang.org/architecture/.

7 Ember.js – A framework for ambitious web developers. Accessed 26.10.2022. URL: https://emberjs.com/.

8 Mohamed Fayad and Douglas C Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997.

9 Richard Feldman. *Elm in Action.* Manning, 2020.

10 Flutter – Build apps for any screen. Accessed 27.01.2023. URL: https://flutter.dev/.

11 Charles Gabriel Foust. *Guaranteeing Responsiveness and Consistency in Dynamic, Asynchronous Graphical User Interfaces.* PhD thesis, Texas A&M University, 2016.

12 Gabriel Foust, Jaakko Järvi, and Sean Parent. Generating reactive programs for graphical user interfaces from multi-way dataflow constraint systems. *SIGPLAN Not.*, 51(3):121–130, October 2015. doi:10.1145/2936314.2814207.

13 Martin Fowler. *Domain-specific languages.* Pearson Education, 2010.

14 Simon Fowler. Model-view-update-communicate: Session types meet the Elm architecture. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, volume 166 of *LIPIcs*, pages 14:1–14:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ECOOP.2020.14.

15 John Freeman, Jaakko Järvi, and Gabriel Foust. HotDrink: A library for web user interfaces. *SIGPLAN Not.*, 48(3):80–83, 2012. doi:10.1145/2480361.2371413.

16 John Alexander Freeman. *Reusable User Interface Behaviors.* PhD thesis, Texas A&M University, 2016.

17 Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1 edition, 1994.

18 Victor Gaudioso. MVVM: Model-view-viewmodel. In *Foundation Expression Blend 4 with Silverlight*, pages 341–367. Apress Berkeley, CA, 2010. doi:10.1007/978-1-4302-2974-2_10.

19 Danny Groenewegen, Zef Hemel, and Eelco Visser. Separation of concerns and linguistic integration in WebDSL. *IEEE Software*, 27(5):31–37, 2010.

20 Danny M. Groenewegen, Zef Hemel, Lennart C.L. Kats, and Eelco Visser. WebDSL: A domain-specific language for dynamic web applications. In *Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, OOPSLA Companion '08, pages 779–780, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1449814.1449858.

21 Magne Haveraaen and Jaakko Järvi. Semantics of multiway dataflow constraint systems. *Journal of Logical and Algebraic Methods in Programming*, 121:100634, December 2020. doi:10.1016/j.jlamp.2020.100634.

22 Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob N. Smith. Property models: From incidental algorithms to reusable components. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, GPCE '08, pages 89–98, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1449913.1449927.

23 JSX. Accessed 26.10.2022. URL: https://facebook.github.io/jsx/.

24 Tim Kadlec. The cost of JavaScript frameworks, April 2020. Accessed 26.10.2022. URL: https://timkadlec.com/remembers/2020-04-21-the-cost-of-javascript-frameworks/.

25 Knockout. Accessed 26.10.2022. URL: https://knockoutjs.com/.

26 Knockout: Observables. Accessed 26.10.2022. URL: `https://knockoutjs.com/documentation/observables.html`.

27 Magnus Madsen, Ondrej Lhotak, and Frank Tip. A semantics for the essence of React. In *European Conference on Object-Oriented Programming*, 2020.

28 Joshua Nussbaum. The Svelte compiler: How it works, February 2020. Accessed 26.10.2022. URL: `https://dev.to/joshnuss/svelte-compiler-under-the-hood-4j20`.

29 Mike Potel. MVP: Model-view-presenter: The Taligent programming model for C++ and Java, 1996. URL: `www.wildcrest.com/Potel/Portfolio/mvp.pdf`.

30 React – A JavaScript library for building user interfaces. Accessed 26.10.2022. URL: `https://reactjs.org/`.

31 Redux – A predictable state container for JavaScript apps. Accessed 27.01.2023. URL: `https://redux.js.org/`.

32 Trygve Reenskaug. Models-views-controllers. Reproduced in The original MVC reports by University of Oslo, December 1979. URL: `https://www.duo.uio.no/bitstream/handle/10852/9621/Reenskaug-MVC.pdf`.

33 Michael Sannella. Skyblue: A multi-way local propagation constraint solver for user interface construction. In *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology*, UIST '94, pages 137–146, New York, NY, USA, 1994. Association for Computing Machinery. `doi:10.1145/192426.192485`.

34 SolidJS – Reactive JavaScript Library. Accessed 26.10.2022. URL: `https://www.solidjs.com/`.

35 State and lifecycle. Accessed 26.10.2022. URL: `https://reactjs.org/docs/state-and-lifecycle.html`.

36 Knut Anders Stokke, Mikhail Barash, and Jaakko Järvi. Manipulating GUI structures declaratively. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2020, pages 63–69, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3425898.3426956`.

37 Knut Anders Stokke, Mikhail Barash, and Jaakko Järvi. A domain-specific language for structure manipulation in constraint system-based GUIs. *Journal of Computer Languages*, 74:101175, 2023.

38 Svelte – Cybernetically enhanced web apps. Accessed 26.10.2022. URL: `https://svelte.dev/`.

39 SwiftUI – Apple Developer Documentation. Accessed 27.01.2023. URL: `https://developer.apple.com/documentation/swiftui/`.

40 Bruce Tate and Curt Hibbs. *Ruby on Rails: Up and Running.* O'Reilly Media, Inc., 2006.

41 Mohit Thakkar. *Building React Apps with Server-Side Rendering.* APress, Berlin, Germany, 1 edition, April 2020.

42 Thinking in React. Accessed 26.10.2022. URL: `https://reactjs.org/docs/thinking-in-react.html`.

43 Two-way binding. Accessed 26.10.2022. URL: `https://angular.io/guide/two-way-binding`.

44 Virtual DOM and internals. Accessed 26.10.2022. URL: `https://reactjs.org/docs/faq-internals.html`.

45 Vue.js – The progressive JavaScript framework. Accessed 26.10.2022. URL: `https://vuejs.org/`.

46 Irena Petrijevcanin Vuksanovic and Bojan Sudarevic. Use of web application frameworks in the development of small applications. In *2011 Proceedings of the 34th International Convention MIPRO*, pages 458–462. IEEE, 2011.

47 Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. *SIGPLAN Not.*, 35(5):242–252, May 2000. `doi:10.1145/358438.349331`.

48 Allen Wirfs-Brock and Brendan Eich. JavaScript: The first 20 years. *Proc. ACM Program. Lang.*, 4(HOPL), June 2020. `doi:10.1145/3386327`.