University of Bergen
Department of Informatics

Multiway Dataflow Constraint System and UI Programming

Author: Mathias Skallerud Jacobsen
Supervisors: Mikhail Barash and Jaakko Järvi

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

June, 2022

Contents

Chapter 1

Introduction

Graphical User Interfaces (GUI) are everywhere and are important for end users. GUIs connect users to software, and often are perceived as *software*. GUIs often have input fields of various kinds, such as text boxes, checkboxes, drop-down lists, and so on. All of these different types of input fields oftentimes lead to complex dependencies between the fields and *event handlers*. An event is an action that takes place when the user interacts with the GUI. A handler is a *callback* that is executed when the event happens. A callback is a function that is passed to another function as an argument, with the intension to be invoked inside of the outer function.

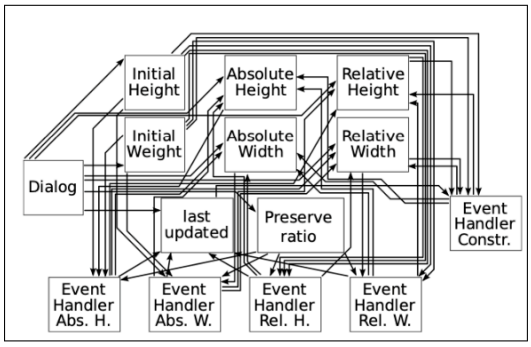

Figure 1.1: Dependencies between widgets and event handlers.

Consider an example of a graphical editing software, such as Adobe Illustrator[1]. There, a user can manipulate the following values: absolute height, absolute width, relative height, relative width, each represented by a text box, and a Boolean indicator whether the ratio must be preserved. A change to any of these input fields must result in corresponding changes of other fields: indeed, for instance if the Boolean indicator is set to true, any changes to the width or height fields will be reflected to opposite field. The dependencies of such program are illustrated in Figure 1.1.

GUIs are often incorrect, to the point where the user often expect them to be wrong. In Figure 1.2, one can see a simple GUI form within a questioner. The error message is displayed, although the date on the from is within the specified boundaries.

[1] `https://www.adobe.com/no/products/illustrator.html`

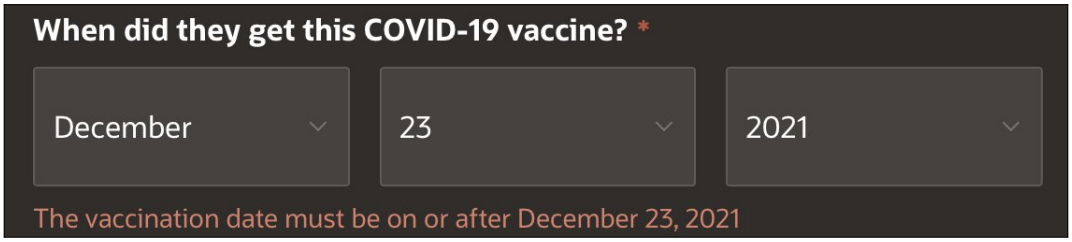

Figure 1.2: Example of a wrong validator.

The new system used by several universities in Norway, made by Direktoratet for forvaltning og økonomistyring (DFØ), has ambiguously formulated question throughout the application. These questions make the application difficult to use, and in certain circumstances, the user is unsure what to click on. One of these questions is presented in the Figure 1.3—however, it is unclear what will happen if either of the buttons is pressed.

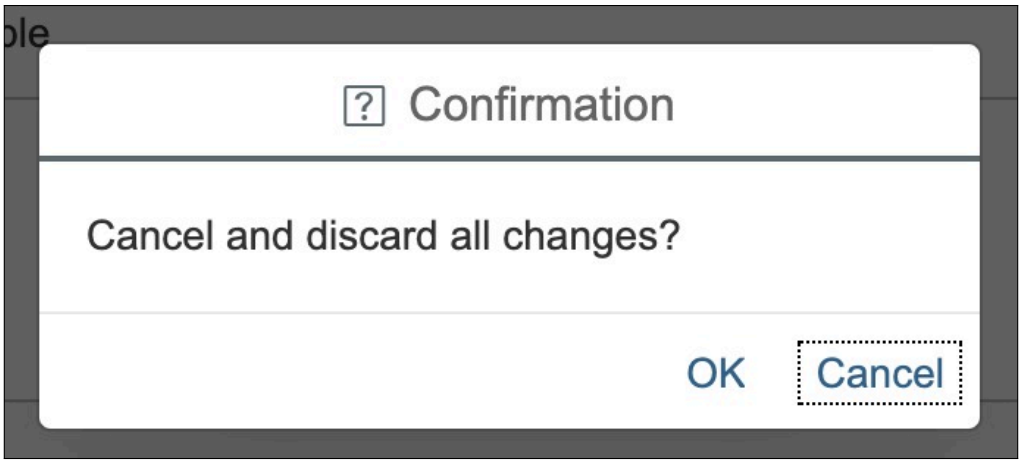

Figure 1.3: Example of an ambiguously formulated question in a GUI.

There are generally a lot of dependencies between all the fields in GUIs with forms that users must fill out. As a result, they are more prone to make errors. One example of this is forms that has mandatory fields; this frequently leads to mistakes in field dependencies. In a form with several phases, one may be instructed to complete all required information before the form allows one to go back a step. This was the case in the registration form for Programming Languages Design and Implementation (PLDI) conference—shown in Figure 1.4. The user had to complete all required fields in the current phase before the GUI enabled the user to return to earlier phases.

Another example of errors in forms is when the validation has a undesirable behavior. Consider, for instance, in the GUI for applying for Finnish citizenship, with a form for writing down every travel abroad, as shown in Figure 1.5. Due to incorrect validation, the user is unable to erase the first trip. Even though the values are entered in the input

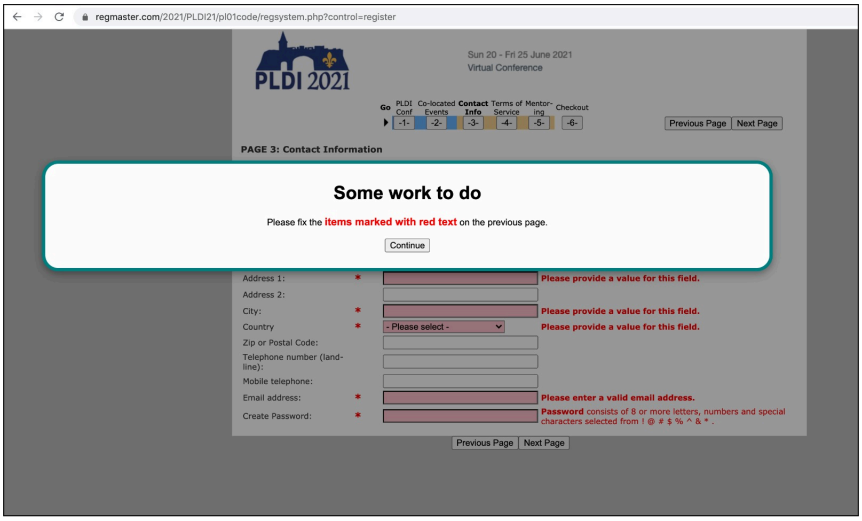

Figure 1.4: Registration for a premier conference in programming languages

fields related to the first trip, a validation error is shown, informing the user that the data is incorrect. Therefore, the user has to erase the entire from, and start all over again.

As previously discussed, GUIs are usually buggy and incorrect, nearly to the point where *users* expect errors. In a relatively modest application, dataflow across fields becomes complicated for the programmer to implement correctly. The way GUIs are developed today, with imperative event handlers that are ran on every user interaction [11], contributes to this. A single error in one of these event handlers might cause a chain of faults in other event handlers. A possible solution is to allow a constraint system to impose dependencies. The constraint system handles the logic behind the dataflow, and the programmer is solely responsible for binding the values in the GUI to the corresponding value in the constraint system. The dependency graph is simplified by allowing the constraint system manage the dependencies—Figure 1.6 is identical to Figure 1.1, except with the use of a constraint system.

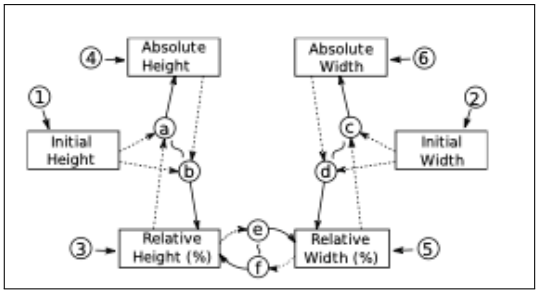

Figure 1.6: Example of dependency graph with a constraint system

The goal of the present project is to understand event-based GUI programming and the limitation to such programming, specify dataflow constraints and understand how they connect to GUI elements, understand the possibilities and limitations of constraint systems based on GUIs, and create an application that reflects this. The multiway dataflow constraint system used in the development of this project is HotDrink [7, 8].

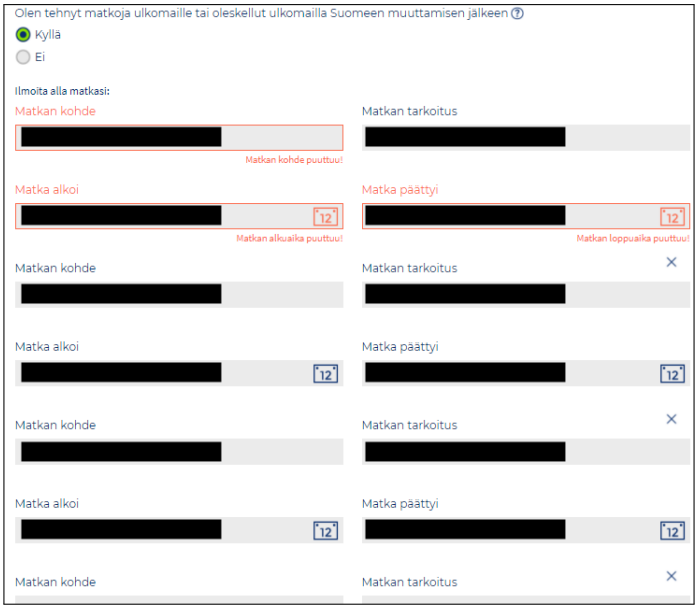

Figure 1.5: System for applying for Finnish citizenship

Before developing this project, I had no prior experience with the constraint system HotDrink.

We have started the implementation of an Integrated Development Environment (IDE) which is aware of the HotDrink language. This will be discussed in more detail in Chapter 4. Hopefully, this will help to the wider adoption of the library.

This report is part of INF319 project at the University of Bergen (UiB).

Chapter 2

Multi-way dataflow constraint systems

A constraint system can be as a tuple $\langle V, C \rangle$, where V is a set of *variables* and C a set of *constraints* [9]. Each variable in V has an associated value of a given type (string, integer, boolean, object, etc.). Each constraint in C is a tuple $\langle R, r, M \rangle$. The variables involved in the constraint is given by $R \subseteq V$, r is some n-ary relation among variables in R, where $n = |R|$. M is a set of non-empty set of *constraint system methods*. Executing any method m in M enforces the constraint by computing values for some subset of R, using another disjoint subset of R as inputs, such that the relation R is satisfied. We denote the inputs and outputs of the method m: *ins(m)* and *outs(m)*. The programmer must ensure that the execution of a method ensures that a constraint is satisfied—the code of a method is to be considered as a "black box".

A constraint is a maintainable relation between a sub-set of the component variables. It can, for instance, represent a equation, such as $E_k = \frac{1}{2}mv^2$ [12]. When a variable changes, the constraint system needs to be re-enforced. The system decides on what constraint satisfaction method, or just method, to execute to enforce the constraint.

The program's *dataflow* is a graph of variables, with directed edges between variables expressing the flow of data or variable dependencies [10]. If variable a is used in the computation of a variable b, then b must be updated whenever a is changed, thus b is said to be dependent on a, and vice versa.

2.1 HotDrink

In this project we used the multiway dataflow constraint system library HotDrink [7, 8], which is a JavaScript-based library for multiway dataflow constraint systems in GUIs. Instead of writing explicit event handlers, the programmer writes declarative specification of data dependencies, from which the library derives the GUI behavior. This library features a domain-specific language (DSL) for defining constraint systems. The DSL allows one to specify *components*, *constraints*, *methods* and *variables*.

In HotDrink, a variable contains a value, and the programmer is able to update this value `v` with a new value `newValue` by calling the `v.set(newValue)` method. To be notified whenever `v` changes, the programmer can subscribe to `v` by calling the `v.subscribe(callback)` method, where the `callback` is a callback function that is called with the new value of `v` whenever `v` changes. There must be established a binding between the variable and the GUI element, so either the GUI is updated when the variable updates, or the variable is updated when the GUI is updated. A variable declaration in HotDrink DSL is similar to variable declaration in JavaScript. The declaration of `v` with a initial value of 1 and w with a initial value of `undefiend` goes as follows:

```
var x = 1, w;
```

The keyword `var` is followed by the name of the variable. The value is optional, and if not specified, the variable is initialized to `undefined`.

In HotDrink, a constraint is a set of variables specified by the constraint satisfaction methods. If we denote the constraint and set of variables by c and vs, respectively, then the constraint satisfaction methods of c enforces c when it is executed. Each method m has a specification that uses two subsets of vs: $ins(m) \subset vs$, $outs(m) \subset vs$ and $ins(m) \cap outs(m) = \emptyset$. It uses the $ins(m)$ as argument(s) to the method, and returns the value(s) that method computes to $outs(m)$. In computer science terms, we state that we read from $ins(m)$ and write to $outs(m)$.

A component is a collection of variables and constraints that forms an independent constraint system [12]. A component typically correspond to a group of GUI elements. Each variable and constraint must be defined inside a component, and by that they are also owned by that component.

Variables often depend on each other, and in that case it gets into a setting of multi-way dataflow. The HotDrink DSL is implemented JavaScript tagged template literals[1], which can be seen in Listing 2.1. This lets the programmer integrate the library with frontend JavaScript frameworks such as React[2] and Svelte[3].

Listing 2.1: Example of the HotDrink DSL

```
import { component } from 'hot-drink';

const comp = component`
    var f=1337, c;

    constraint c1 {
        m1(c -> f) => c * (9/5) + 32;
        m2(f -> c) => (f -32) * 5/9;
    }
`;
```

Currently, there are no integrated methods in HotDrink to bind the constraint system to frontend frameworks. The programmer has to decide on the best way to integrate the HotDrink to the frontend web application of there choosing. Listings 2.2, 2.3, 2.4 show one way of binding HotDrink with Svelte.

Listing 2.2: Function for binding HotDrink and Svelte variable

```
function setHDValue<T>(HDvariable: Variable<T>, n: T) {
    if (n !== HDvariable.value) {
        HDvariable.set(n);
    };
};
```

Listing 2.3: Using the reactive statement, in Svelte [2], to update HotDrink corresopnding value, and trigger HotDrink to enforce the constraint system

```
$: {
    setHDValue(HotDrinkValue, frontendValue);
}
```

Listing 2.4: Using the onMount callback, in Svelte [2], to update the frontend value that correspont to the same value in HotDrink, when the HotDrink value changes.

```
onMount(() => {
    HotDrinkValue.subscribeValue((value: number) => frontendValue = value);
});
```

In the current sate of the library there is no developer-friendly documentation for the library. This makes it hard for new developers to adopt the library. There are currently two ways writing HotDrink code: the developer can either access the HotDrink

[1] Template literals
[2] https://reactjs.org/
[3] https://svelte.dev/

Application Programming Interface (API) seen in listing 2.5, or the developer can use the HotDrink DSL seen in listing 2.1. Both of the approaches have their own set of disadvantages.

Currently, HotDrink does not have a dedicated tool support—i.e., IDE—which would provide DSL-level syntax highlighting, code validation, refactoring and debugging. Using the API can easily lead to errors and is not recommended. On line 9 and 10 of listing 2.5 the developer have to specify the index of the input-variables and the output-variables. In the example there are only two variables—`c` and `f`—thus it is not complicated to keep track of the index for the variables. In lager systems, it is easy to get lost in all the variables and their indexes.

Using the DSL is the most used way to write HotDrink code. This is also easier to understand and more compact code then the API, but the developer has to spend time learning the DSL. The difference can be seen in the Listings 2.1 and 2.5. Here both have the same *temperature conversion* example. In this project we used the HotDrink DSL. The use of the tagged template literals in the DSL, as discussed, makes the programming process error-prone. Finding and fixing code errors becomes a significantly difficult process. In systems larger than the temperature conversion example, it is easy to get lost in all the constraints, not knowing which variables are affected by other constraints etc. This may lead to a non-desirable behavior in the system. This together with the fact that the error messages from HotDrink when using the DSL are not helpful to the developer. There are no way of differentiating between a missing semi-colon or a missing closing bracket. The error messages are generic error stacks.

Listing 2.5: Example of how to use the HotDrink API to simulate the relationship between fahrenheit and celsius.

```
1  import { Component, Method, ConstraintSpec, MaskNone } from 'hotdrink';
2
3  // create a component and emplace some variables
4  export const Temp = new Component("Temp");
5  const c = Temp.emplaceVariable("c", 1337);
6  const f = Temp.emplaceVariable("f");
7
8  // create a constraint spec
9  const toFahrenheit = new Method(2, [0], [1], [MaskNone], (c) => (c * 9 / 5
       ↪ + 32));
10 const toCelsius = new Method(2, [1], [0], [MaskNone], (f) => (f - 32) * 5
       ↪ / 9);
11
12 const graderSpec = new ConstraintSpec([toFahrenheit, toCelsius]);
13
14 // emplace a constraint built from the constraint spec
15 const grader = Temp.emplaceConstraint("grader", graderSpec, [c, f]);
```

Chapter 3

Implementation of a parcel sending form

3.1 Posten

The Norwegian post-service handles packages domestically and internationally from and to Norway. A package can in many cases be seen as a box. Thus, we decided to make a web application that would calculate the price for sending a package with Posten. To still keep the application simple, we decided to only support packages sent domestically. Posten measures the height, depth, width and weight of the package, this is used to calculate the price of sending the package. Posten uses the price model[1] seen in table 3.1 for domestic packages [1].

Listing 3.1: HotDrink logic for determining the price

```
constraint price {
    m1(w, d, h, kg -> price) => {
        if (kg <= 5) {
            if (h <= 35 && w <= 25 && d <=12) {
                return 70;
            } else if (h <= 120 && w <= 60 && d <=60) {
                return 129;
            } else {
                let spesialgodstillegg = 149;
                return 129 + spesialgodstillegg;
            }
        } else if (kg <=10) {
            if (h <= 120 && w <= 60 && d <=60) {
                return 129;
            } else {
```

[1] If any of the measures — height, depth, width — of the package is larger than the maximum allowed there is a special package fee of 149 NOK.

```
                let spesialgodstillegg = 149;
                return 129 + spesialgodstillegg;
            }
        } else if (kg <= 25) {
            if (h <= 120 && w <= 60 && d <=60) {
                return 229;
            } else {
                let spesialgodstillegg = 149;
                return 229 + spesialgodstillegg;
            }
        } else if (kg <= 35) {
            if (h <= 120 && w <= 60 && d <=60) {
                return 299;
            } else {
                let spesialgodstillegg = 149;
                return 299 + spesialgodstillegg;
            }
        }
    }
}
```

Table 3.1: Pricing model for domestic packages from Posten

	<35 x 25 x 12 cm	<120 x 60 x 60 cm
0-5 kg	70 NOK	129 NOK
0-10 kg	129 NOK	
10-25 kg	229 NOK	
25-35 kg	299 NOK	

3.2 Svelte

In this project, to implement the application, we use the Svelte. Svelte is a JavaScript framework used for building user interfaces, just like React, Angular, Vue, etc. Where some of these frameworks bulk there work in the browser, Svelte shifts that work onto a compile step [3]. Svelte is written using TypeScript. Instead of using virtual Document Object Model (DOM) Svelte uses build time to covert the code into JavaScript [2].

3.3 ThreeJS

For better usability of the tool, the box is rendered in 3D using JavaScript library ThreeJS [5]. We use SveltThree [4] which is a component library written for Svelte using ThreeJS.

SveltThree works by adding *scene*s to an *canvas* component. A scene could include a *camera*, *lights* and *meshes*. Meshes are used to represent a object in the scene—the box or

package in our case—and the camera where the view of the canvas is placed. The different types of light are there to make the object feel more natural, in the application we use two types of lights: *ambient-* and *directional-light*. The canvas also takes a *WebGLRender* component which is used to add configurations to the renderer. The setup for SveltThree usage in our application can be seen in Listing 3.2.

Listing 3.2: Example of a SveltThree setup

```
<Canvas let:sti w={canvasWidth*.6} h={canvasHeight * .6} interactive>

    <Scene {sti} let:scene id="scene1" props={{ background: 0x00000 } } >

        <PerspectiveCamera {scene} id="cam1" pos={[0, 0, 0]} lookAt={[0,
            ↪ 0, 0]} />
        <AmbientLight {scene} intensity={1.25} />
        <DirectionalLight {scene} pos={[3, 3, 3]} />
        <Mesh
        {scene}
        geometry={cubeGeometry}
        material={cubeMaterial}
        mat={{ roughness: 0.5, metalness: 0.5, color: 0xFF8001, }}
        pos={[-1, 0, 0]}
        rot={[.3, .4, 0]}
        scale={[1, 1, 1]}
        />

        <Mesh
            {scene}
            geometry={sphereGeometry}
            material={sphereMaterial}
            mat={{ roughness: 0.5, metalness: 0.5, color: 0xF6E05E, }}
            pos={[$widthS*(3/4)+ ($depthS/5), 0, 0]}
            rot={[.2, .2, 0]}
            scale={[1, 1, 1]}
        />

    </Scene>

    <WebGLRenderer
    {sti}
    sceneId="scene1"
    camId="cam1"
    config={{ antialias: true, alpha: true }}
    />

</Canvas>
```

Chapter 4

Discussion

Graphical user interfaces are difficult to implement correctly, and often lead to bugs and a cluster of dependencies. For a long time, this has been the case, to the point that the user now expects the GUI to have errors. These issues are frequently caused by the fact that current GUI programmers manage all user interactions using event driven programming paradigms [7]. Due to the asynchronous nature of this approach, it is difficult to ensure that the same result is obtained from the same amount of sequences, but at different times. By letting the constraint system handle the coordinating asynchronous updates, we release the programmer from this error-prone task. The constraint system updates the GUI, even when some calculations fail, the resultant reactive program assures consistent outputs for any sequence of editing actions, despite the numerous conceivable interleaving of events and computations.

I have discovered how a constraint system, such as HotDrink, might be a useful alternative to event-driven programming. Constraint systems make it easier to deal with significant changes in business logic, along with more readable code. I have implemented a parcel sending form for parcels sent by the Norwegian post service domestically. Compared to Posten's own solution, our approach features; input-fields for the dimensions,*height, width, depth*, the *weight* and *volume* of the package and the *price* for sending the package. As an extra feature, the user are also able to see the package in the given dimensions in a 3D-view, in our application. Posten itself have a website for doing this, but instead of letting the user define their own box, there are a form with standard set sizes that the user can select. When the user have selected all the required checkboxes the price for that collection of selected checkboxes is shown. The 3D-view also contains a tennis-ball, with a radius of 3.4, as a real life comparison object. A change in any of the dimensions, height, width or depth, will cause a change to the volume etc.

HotDrink can be integrated with various web frameworks, such as React, Svelte, etc. When using the library the developer, as discussed, is expected to know the syntax of the DSL embedded into HotDrink, and this might prevent wider adoption of the library. To mitigate this, we are currently developing tool support for HotDrink. The development tool is intended to be a implementation of the HotDrink DSL as a plugin for Visual Studio Code. This plugin should support IDE features mentioned in Section 2.1.We intend to pay a special attention to implementing the debugger functionality, such as the dataflow in a constraint system. We also intend to look at various views and visualization of running constraint systems solver, such as:

- showing the current variable values in the constraint system;
- showing the current dataflow, with which we mean to highlight which method in the dataflow is currently being executed;
- showing the history of previous dataflows;
- showing the generation graph, to visualize the entire history of values and how certain values have been used to compute new values;
- showing whether the constraints are active or not.

We use the language workbench Eclipse Xtext [6] / Langium to implement the DSL. From this DSL we get syntax highlighted *keywords*, and also autocompletion of these keywords.

Bibliography

[1] Sende brev og pakker - Posten Norge.
URL: https://sending.posten.no/bestill/pakke?_ga=2.21151167.568534240.1651753510-2089638552.1651753510. [Accessed on *2022-05-23*]. Send brev eller Norgespakke med Posten.

[2] Docs • Svelte.
URL: https://svelte.dev/docs. [Accessed on *2022-05-05*].

[3] Svelte • Cybernetically enhanced web apps.
URL: https://svelte.dev/. [Accessed on *2022-05-23*].

[4] svelthree • Svelte powered three.js development.
URL: https://svelthree.dev. [Accessed on *2022-05-05*]. An open source Svelte components library for declarative construction of reactive and reusable Three.js scene graphs. Developed by Vatroslav Vrbanic.

[5] Three.js – JavaScript 3D Library.
URL: https://threejs.org/. [Accessed on *2022-05-23*].

[6] Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion - SPLASH '10*, page 307, Reno/Tahoe, Nevada, USA, 2010. ACM Press. ISBN 9781450302401. doi: 10.1145/1869542.1869625.
URL: http://portal.acm.org/citation.cfm?doid=1869542.1869625.

[7] Gabriel Foust, Jaakko Järvi, and Sean Parent. Generating reactive programs for graphical user interfaces from multi-way dataflow constraint systems. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2015, page 121–130, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336871. doi:

10.1145/2814204.2814207.
URL: https://doi.org/10.1145/2814204.2814207.

[8] John Freeman, Jaakko Järvi, and Gabriel Foust. Hotdrink: A library for web user interfaces. *SIGPLAN Not.*, 48(3):80–83, sep 2012. ISSN 0362-1340. doi: 10.1145/2480361.2371413.
URL: https://doi.org/10.1145/2480361.2371413.

[9] Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob Smith. Algorithms for user interfaces. In *Proceedings of the eighth international conference on Generative programming and component engineering - GPCE '09*, page 147, Denver, Colorado, USA, 2009. ACM Press. ISBN 9781605584942. doi: 10.1145/1621607.1621630.
URL: http://portal.acm.org/citation.cfm?doid=1621607.1621630.

[10] Knut Anders Stokke. Declaratively programming the dynamic structure of graphical user interfaces. Master's thesis, The University of Bergen, 2020.

[11] Knut Anders Stokke, Mikhail Barash, Karl Henrik Elg Barlinn, Daniel Berge, Torjus Schaathun, and J Jaakko. Multi-way dataflow specifications in graphical user interfaces. In *Norsk IKT-konferanse for forskning og utdanning*, number 1, pages 161–163, 2021.

[12] Rudi Blaha Svartveit. Multithreaded multiway constraint systems with rust and webassembly. Master's thesis, The University of Bergen, 2021.