

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

May, 2022

Contents

1	Introduction	1
2	Multi-way dataflow constraint system	6
2.1	HotDrink	7
3	Implementation of a parcel sending form	10
3.1	Posten	10
3.2	Svelte	11
3.3	ThreeJS	11
3.4	Early implementation	12
4	Discussion	14
4.1	Current work on HotDrink	14
	List of Acronyms and Abbreviations	15
	Bibliography	16

List of Figures

1.1	Dependencies between widgets and event handlers.	1
1.2	Example of a wrong validator	2
1.3	Example of an ambiguously formulated question	2
1.4	Registration for a premier conference in programming languages	3
1.5	System for applying for Finnish citizenship	3
1.6	Finnish tax system	4
1.7	Example of dependency graph with a constraint system	4

List of Tables

3.1 Pricing model for domestic packages from Posten	11
---	----

Listings

2.1	Example of the HotDrink Domain Specific Language (DSL)	8
2.2	Function for binding HotDrink and Svelte variable	8
2.3	Using the reactive statement, in Svelte [2], to update HotDrink corresponding value, and trigger HotDrink to enforce the constraint system . . .	8
2.4	Using the onMount callback, in Svelte [2], to update the frontend value that correspond to the same value in HotDrink, when the HotDrink value changes.	8
2.5	Example of how to use the HotDrink Application Programming Interface (API) to simulate the relationship between fahrenheit and celsius	9
3.1	HotDrink logic for determining the price	10
3.2	Example of a SveltThree setup	12
3.3	Example of the constraint calculating the values of the box	13

Chapter 1

Introduction

Graphical User Interface (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. Consider an example of a graphical editing software, such as Adobe Il-

lustrator¹. The dependencies of such program are illustrated in Figure 1.1. 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.

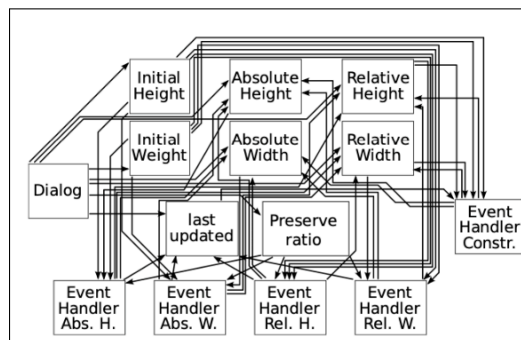


Figure 1.1: Dependencies between widgets and event handlers.

In Figure 1.2, you can see a simple GUI form within a questioner. The error message is displayed, although the date on the form is within the specified boundaries.

The new system used by several universities in Norway, made by Direktoratet for forvaltning og økonomistyring (DFØ), has ambiguously formulated question throughout

¹<https://www.adobe.com/no/products/illustrator.html>

When did they get this COVID-19 vaccine? *

December 23 2021

The vaccination date must be on or after December 23, 2021

Figure 1.2: Example of a wrong validator

Confirmation

Cancel and discard all changes?


OK Cancel

Figure 1.3: Example of an ambiguously formulated question

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.

There are generally a lot of dependencies between all the fields in GUIs with forms that the user 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, you may be instructed to complete all required information before the form allows you 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. For instance, in a GUI with a form for writing down every travel abroad, such seen in Figure 1.5. Due to incorrect validation, the user is unable to erase the first trip. Even though the details provided in the first trip, a validation error is shown, informing the user that the data is incorrect. Therefore, the user have to erase the the hole from, and start all over again.

← → ↻ **regmaster.com/2021/PLDI21/pldi01code/regsystem.php?control=register**



Sun 20 - Fri 25 June 2021
Virtual Conference

Go

PLDI Conf Events Info Service Mentoring

Co-located

Contact

Terms of Mentoring

Checkout

1

2

3

4

5

6

Previous Page

Next Page

PAGE 3: Contact Information

Some work to do

Please fix the **Items marked with red text** on the previous page.

Continue

Address 1:

*

Please provide a value for this field.

Address 2:

*

Please provide a value for this field.

City:

*

Please provide a value for this field.

Country:

*

- Please select -

Please provide a value for this field.

Zip or Postal Code:

*

Please provide a value for this field.

Telephone number (landline):

*

Please provide a value for this field.

Mobile telephone:

*

Please provide a value for this field.

Email address:

*

Please enter a valid email address.

Create Password:

*

Password consists of 8 or more letters, numbers and special characters selected from ! @ # \$ % ^ & * .

Previous Page

Next Page

Figure 1.4: Registration for a premier conference in programming languages

Figure 1.5: System for applying for Finnish citizenship

Olen tehnyt matkoja ulkomaille tai oleskellut ulkomailta Suomeen muuttamisen jälkeen [?](#)

☒ Kyllä
☐ Ei

Ilmoita alla matkasi:

<p>Matkan kohde</p> <div> <div></div> <div>Matkan kohde puuttuu!</div> </div>	<p>Matkan tarkoitus</p> <div> <div></div> </div>
<p>Matka alkoi</p> <div> <div></div> <div>12</div> <div>Matkan alkuaika puuttuu!</div> </div>	<p>Matka päättyi</p> <div> <div></div> <div>12</div> <div>Matkan loppuaika puuttuu!</div> </div>
<p>Matkan kohde</p> <div> <div></div> </div>	<p>Matkan tarkoitus</p> <div> <div></div> <div>×</div> </div>
<p>Matka alkoi</p> <div> <div></div> <div>12</div> </div>	<p>Matka päättyi</p> <div> <div></div> <div>12</div> </div>
<p>Matkan kohde</p> <div> <div></div> </div>	<p>Matkan tarkoitus</p> <div> <div></div> <div>×</div> </div>
<p>Matka alkoi</p> <div> <div></div> <div>12</div> </div>	<p>Matka päättyi</p> <div> <div></div> <div>12</div> </div>
<p>Matkan kohde</p> <div> <div></div> </div>	<p>Matkan tarkoitus</p> <div> <div></div> <div>×</div> </div>

Henkilösuhteet	
Lapset ⓘ	
Avoliitto	
Puolison nimi	null, null
Rekisteröity parisuhde	
Puolison nimi	null, null
Vanhemmat ⓘ	
Edunvalvontavaltuus ⓘ	
Henkilöedunvalvontavaltutettu	null, null

Figure 1.6: Finnish tax system

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. 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. Allowing a con-

straint system to impose dependencies is one solution. 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 1.7 is identical to Figure 1.1, except with the use of a constraint system.

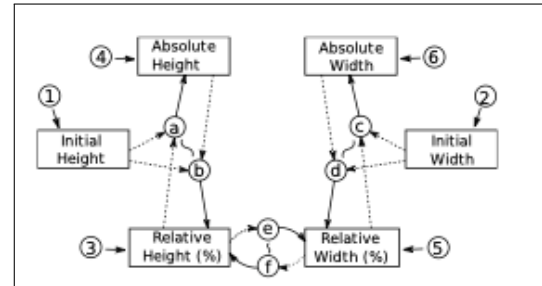


Figure 1.7: Example of dependency graph with a constraint system

The goal of the project was 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. Finally, create an application with this in mind. The multiway dataflow constraint system used in the development of this project is HotDrink [8]. Before developing this project I had no prior experience with the constraint system HotDrink.

For the later point, 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

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

Chapter 2

Multi-way dataflow constraint system

A constraint system is seen 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 like to 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".

constraint satisfaction methods...

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 programs *dataflow* is a graph of variables, with directed edges between variables expressing data flow 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], 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 *undefined* goes as follows:

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

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

A constraint, in HotDrink, is a set of variables specified by the constraint satisfaction methods. If we denote the constraint and set of variables to 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 arguments to the method, and returns the value that method computes to $outs(m)$. In computer science terms, we state that we read from $ins(m)$ and writes 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, 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 React² and Svelte³.

Listing 2.1: Example of the HotDrink DSL

```

1 import { component } from 'hot-drink';
2
3 const comp = component`
4   var f=1337, c;
5
6   constraint c1 {
7     m1(c -> f) => c * (9/5) + 32;
8     m2(f -> c) => (f -32) * 5/9;
9   }
10 `;

```

Currently there are no integrated methods in HotDrink to bind the constraint system to frontend frameworks. The programmer have 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 shows one way of binding HotDrink with Svelte.

Listing 2.2: Function for binding HotDrink and Svelte variable

```

1 function setHDValue<T>(HDvariable: Variable<T>, n: T) {
2   if (n !== HDvariable.value) {
3     HDvariable.set(n);
4   };
5 };

```

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

```

1 $: {
2   setHDValue(HotDrinkValue, frontendValue);
3 }

```

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

```

1 onMount(() => {
2   HotDrinkValue.subscribeValue((value: number) => frontendValue = value);
3 });

```

In the current sate of the library there is no easy documentation for the library. There are two ways writing HotDrink. The developer can either access the HotDrink Application Programming Interface (API) seen in listing 2.5, or the developer can use the HotDrink DSL seen in listing 2.1. Both with there own disadvantages.

¹Template literals

²<https://reactjs.org/>

³<https://svelte.dev/>

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 larger 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 than the API, as seen in the listing 2.1 and 2.5. Here both have the same *celsius to fahrenheit* example. In this project we used the HotDrink DSL. The use of the tagged template literals, as discussed, makes the programming process error-prone. Finding and fixing code errors becomes a significantly difficult process. In systems larger than the *celsius to fahrenheit*, 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

In this chapter...

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 give you 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. Thus we had to add a new variable to the constraint system — **weight** — see listing 3.1. With it, the logic for calculating the price. Posten uses the price model [1]¹ seen in table 3.1 for domestic packages.

Listing 3.1: HotDrink logic for determining the price

```
1 constraint price {  
2   m1(w, d, h, kg -> price) => {  
3     if (kg <= 5) {  
4       if (h <= 35 && w <= 25 && d <=12) {  
5         return 70;  
6       } else if (h <= 120 && w <= 60 && d <=60) {  
7         return 129;  
8       } else {  
9         let spesialgodstillegg = 149;
```

¹If 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.

```

10         return 129 + spesialgodstillegg;
11     }
12 } else if (kg <=10) {
13     if (h <= 120 && w <= 60 && d <=60) {
14         return 129;
15     } else {
16         let spesialgodstillegg = 149;
17         return 129 + spesialgodstillegg;
18     }
19 } else if (kg <= 25) {
20     if (h <= 120 && w <= 60 && d <=60) {
21         return 229;
22     } else {
23         let spesialgodstillegg = 149;
24         return 229 + spesialgodstillegg;
25     }
26 } else if (kg <= 35) {
27     if (h <= 120 && w <= 60 && d <=60) {
28         return 299;
29     } else {
30         let spesialgodstillegg = 149;
31         return 299 + spesialgodstillegg;
32     }
33 }
34 }
35 }

```

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

Svelte is a JavaScript framework used for building user interfaces, just like React, Angular, Vue. 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 ended up using SveltThree [4] which is a component library written for Svelte using ThreeJS.

SvelteThree works by adding *Scenes* 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 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.

Listing 3.2: Example of a SvelteThree setup

```

1 <Canvas let:sti w={canvasWidth*.6} h={canvasHeight * .6} interactive>
2
3   <Scene {sti} let:scene id="scene1" props={{ background: 0x000000 }} >
4
5     <PerspectiveCamera {scene} id="cam1" pos=[[0, 0, 0]] lookAt=[[0,
6       ↗ 0, 0]] />
7     <AmbientLight {scene} intensity={1.25} />
8     <DirectionalLight {scene} pos=[[3, 3, 3]] />
9     <Mesh
10      {scene}
11      geometry={cubeGeometry}
12      material={cubeMaterial}
13      mat={{ roughness: 0.5, metalness: 0.5, color: 0xFF8001, }}
14      pos=[[ -1, 0, 0]]
15      rot=[[.3, .4, 0]]
16      scale=[[1, 1, 1]]
17    />
18    <Mesh
19      {scene}
20      geometry={sphereGeometry}
21      material={sphereMaterial}
22      mat={{ roughness: 0.5, metalness: 0.5, color: 0xF6E05E, }}
23      pos=[[ $widthS*(3/4)+ ($depthS/5), 0, 0]]
24      rot=[[.2, .2, 0]]
25      scale=[[1, 1, 1]]
26    />
27
28  </Scene>
29
30  <WebGLRenderer
31  {sti}
32  sceneId="scene1"
33  camId="cam1"
34  config={{ antialias: true, alpha: true }}
35  />
36
37 </Canvas>

```

3.4 Early implementation

In the early stages of implementation the idea for the application was to learn how to use the constraint system library HotDrink in a web application. Thus the application in mind was a simple GUI for a box in 3D. Her we wanted to have few constraints and keep the complexity of the constraints low.

In the first iteration of the project the box had three dimensions: *depth*, *width* and *height*. From these three values we calculate the value for the *volume* of the box. See `calculateVolume` method in listing 3.3. `calculateHeight`, `calculateDepth`, `calculateWidth` methods calculate the values for `height`, `depth` and `width` individually, whenever the value of `volume` is updated. Then `HotDrink` checks the *priority list*, and chooses the one that was updated last.

Listing 3.3: Example of the constraint calculating the values of the box

```
1 var width=1, depth=1, height=1, volume;
2
3 constraint metrics {
4   calculateVolume(width, depth, height -> volume) => width * depth *
      ↪ height;
5   calculateHeight(volume, depth, width -> height) => volume / (depth *
      ↪ width);
6   calculateDepth(volume, width, height -> depth) => volume / (width *
      ↪ height);
7   calculateWidth(volume, depth, height -> width) => volume / (depth *
      ↪ height);
8 }
```

`HotDrink` previously been used with `React`, too some success. Thus we decided to test it on `Svelte`. From listings 2.2, 2.3, 2.4 as seen in section 2.1 we can see how to bind `HotDrink` to `Svelte`. This becomes discombobulating when more values are added. Thus when we add one variable in `HotDrink`, we have to make the same corresponding variable in `Svelte`. Then we have to bind these two variables together. This is done in three steps.

First we use the `onMount`-function in `Svelte`, to make the `Svelte` variable subscribe to the `HotDrink` variable using the `subscribeValue`-function. See listing 2.4. This makes it that when `HotDrink` enforces the constraint the `Svelte` variable is updated. The `onMount`-function is used to make sure the subscribe only happens one time, after the component is mounted. Now the changes propagates one way from `HotDrink` to `Svelte`. Thus changes from `Svelte` to `HotDrink` also needs to be propagated. For this to happen we use the *reactive statement* in `Svelte`. See listing 2.3. The reactive statement then call the function `setHDValue`, seen in listing 2.2. This makes it so when a variable is updated in the GUI, the corresponding variable in `HotDrink` is updated.

Chapter 4

Discussion

4.1 Current work on HotDrink

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 ???. 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, as seen in figure ??.

List of Acronyms and Abbreviations

API Application Programming Interface.

DFØ Direktoratet for forvaltning og økonomistyring.

DOM Document Object Model.

DSL Domain Specific Language.

GUI Graphical User Interface.

IDE Integrated Development Environment.

PLDI Programming Languages Design and Implementation.

UiB University of Bergen.

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] svelte-three • Svelte powered three.js development.
URL: <https://svelte-three.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>. HotDrink is a JavaScript library for constructing forms, dialogs, and other common user interfaces for Web applications. With HotDrink, instead of writing event handlers, developers declare a "view-model" in JavaScript and a set of "bindings" between the view-model and the HTML elements comprising the view. These specifications tend to be small, but they are enough for HotDrink to provide a fully operational GUI with multi-way dataflows, enabling/disabling of values, activation/deactivation of commands, and data validation. HotDrink implements these rich behaviors, expected of high-quality user interfaces, as generic reusable algorithms. This paper/tool demonstration introduces developers to the HotDrink library by stepping through the construction of an example web application GUI. The library is a concrete realization of our prior work on the "property models" approach to declarative GUI programming. To encourage adoption among developers, we have packaged the technology following established web programming conventions.

- [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.