

UNIVERSITY OF BERGEN
DEPARTMENT OF INFORMATICS

Visual specification of multi-way data-flow constraint systems

Author: Daniel Berge

Supervisors: Mikhail Barash, Jaakko Järvi



UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

May, 2022

Abstract

User interfaces are costly to develop and difficult to get correct. Estimates place the effort of programming UIs between 30% and 60% of the total programming effort of applications. One reason for why graphical user interface (GUI) programming is difficult is that there are lots of interdependencies between widgets, and they easily get lost in code.

This thesis is motivated by increasing the effectiveness of GUI programming, making the gap between code and behavior smaller, by replacing a lot of the code with a visual diagram that the programmer draws interactively. We are tackling this problem with a declarative programming approach based on multi-way data-flow constraint systems.

This thesis shows that data-flow constraint system based GUI programming makes the visual specification of GUIs feasible and that implementing visual cues to users is cheap in the approach — we show through small experiments that programmers save effort and users benefit. This thesis suggests how to simplify the programming of user interfaces, and get rid of complicated event-handling that most GUI code essentially has.

The visual client developed in this thesis project is built on top of the HotDrink library. HotDrink is a library that is used for writing GUIs declaratively, making multi-way data-flow constraint systems and maintaining dependencies explicit. HotDrink offers a good platform for creating a visual tool; the visual tool can merely edit HotDrink constraints and make them visible to the programmer, and ultimately to the user.

We create a visual tool for creating HotDrink applications. The tool is able to create constraints between widgets, in a visual way, that generates correct constraint systems written in HotDrink. The tool is able to generate complete programs that implement both the Hypertext Markup Language specification (the view) and the HotDrink code (the model).

Acknowledgements

First and foremost, I would like to thank my supervisors Mikhail Barash and Jaakko Järvi for their support and guidance. Their knowledge and experience have been invaluable in the development of this thesis.

I would also like to thank my friends at the university for their help, support and many great conversations over lunch.

Lastly, I would like to thank my family for their support throughout the degree and this thesis.

Daniel Berge

01 May, 2022

Contents

1	Introduction	1
1.1	Thesis outline	2
2	Background	4
2.1	Motivation	4
2.2	Constraint system	6
2.3	HotDrink	7
2.4	Visual programming	8
2.5	Technologies	8
2.5.1	TypeScript	10
2.5.2	React	10
2.5.3	Eclipse Xtext	11
3	A visual language for HotDrink specifications	12
3.1	Form designer	12
3.1.1	Visual representation of constraints	15
3.1.2	Running and exporting	19
3.2	Constraint editor	20
3.2.1	Code view	20
3.2.2	Visual programming interface	21
4	Implementation	25
4.1	Form Designer	25
4.2	Visual programming interface	26
4.3	HotDrink	28
4.3.1	Components and constraints	28
4.3.2	Serialization and deserialization	29
5	Case study — Norwegian tax form	31

6 Usability test	35
6.1 Setup	35
6.2 Results	36
6.2.1 Feedback	37
6.3 Summary	39
7 Related work	40
7.1 Constraint systems	40
7.1.1 Amulet Environment	40
7.1.2 ConstraintJS	41
7.2 Form builders	41
7.2.1 JotForm	42
7.2.2 Microsoft Visual C# Express	42
7.2.3 Delphi	43
7.3 Lowcode / no code environments	43
7.3.1 Pure Data	44
7.4 Visual programming environments	44
7.4.1 LabVIEW	44
7.4.2 Empirical evidence for and against visual programming	45
7.5 Language workbenches	46
7.5.1 Eclipse Xtext	46
7.5.2 Langium	46
7.5.3 Whole platform	47
7.5.4 MetaEdit+	47
8 Conclusion and future work	50
8.1 Conclusion	50
8.2 Future work	50
List of Acronyms	53
Bibliography	54
A DSL standard library	58

List of Figures

1.1	The 3-layer architecture of the visual editor.	1
2.1	A covid vaccination form, where the user is not able to submit valid input.	4
2.2	A form for applying for Finnish citizenship, where the user is not able to remove the first trip. "Matkan kohde puuttuu" means that the destination of the trip is missing, "Matkan alkuaika puuttuu" means that the start time of the trip is missing and "Matkan loppuaika puuttuu" means that the end time of the trip is missing.	5
2.3	Example of a Blockly application, printing "Hello World!" three times [2].	9
2.4	Example of a visual data-flow diagram, adding two numbers together [15].	9
3.1	A form with fields <code>firstname</code> and <code>lastname</code> , with <code>lastname</code> selected (Layer 1).	13
3.2	The properties panel shows the properties of the selected <code>lastname</code> element (Layer 1).	14
3.3	Highlighted elements while in <i>new constraint</i> -mode (Layer 2).	15
3.4	A constraint connected to two variables, with a method with one output variable (Layer 2).	16
3.5	Hovering over a method will highlight the data-flow of the method. Here the effect of hovering over the method <code>Divide</code> is shown (Layer 2).	17
3.6	Hovering over a method will highlight the data-flow of the method. Here the effect of hovering over the method <code>Multiply</code> is shown (Layer 2).	17
3.7	The WHAP constraint system example illustrated as a graph [25].	18
3.8	WHAP example visualized in the visual editor (Layer 2).	19
3.9	An example of <i>run</i> -mode with the implementation of WHAP.	19
3.10	Codeview implementation of converting <code>celsius</code> to <code>fahrenheit</code> , showing autocomplete when writing <code>celcius</code> . (Layer 3).	20
3.11	Visual programming interface calculating the max value of two inputs (Layer 3).	22

4.1	An example method implementation using our visual programming interface built with Rete. This method checks if a number is positive and returns true or false accordingly.	27
5.1	Norwegian tax form using HotDrink, JavaScript and HTML.	31
5.2	Norwegian tax form in the visual editor's design mode.	32
6.1	When clicking on <i>Create method</i> , an input field and two buttons appear.	37
6.2	The top part of the constraint editor.	38
6.3	Visual programming blocks showing sockets between <i>width</i> , <i>height</i> and a multiplication block.	39
7.1	Example of comma separated list of names, using ConstraintJS. The code for this application is shown in Listing 7.1.	42
7.3	Some screenshots of the related works presented in this chapter.	49

List of Tables

6.1	A table of the results of the usability test. Green (●) means that the participant completed the task successfully. Yellow (◐) means that the participant completed it, but it either took a long time or multiple tries. Red (◑) means that the participant failed to complete the task and we had to intervene to help the participant complete the task. White (-) means that the task was skipped.	36
A.1	Visual blocks available in the standard DSL library (Layer 3).	58

Listings

2.1	Temperature converter using HotDrink’s DSL.	7
2.2	A counter example implemented in React with hooks [5]. In React, hooks can be created to define state in the application; here we use a hook to hold the counter value. Using <code>React.useState</code> we get a value and a function to set the value. We define our own increment function to increment the count, and call this when the user clicks the button. The incremented value will then be updated in the JSX tree (JSX is React’s way of defining HTML and JS together).	10
3.1	WHAP implemented in HotDrink.	18
3.2	Generated code output of maximum value example. The first line finds the maximum of the two input variables, and the second line returns the maximum value (Layer 3).	21
3.3	The grammar of the DSL in EBNF (Layer 3).	23
3.4	Example of a visual block implemented using the DSL (Layer 3).	23
3.5	Contains example of a visual block using the DSL (Layer 3).	23
4.1	Storable format of temperature converter example in JSON.	30
5.1	Norwegian tax form constraints in HotDrink, calculating tax and income, with deduction.	33
7.1	A ConstraintJS example, listing commander names in a list [35]. The following is a template that is connected to the ConstraintJS constraint system. The DOM elements are bound to the constraint system, such that when a change occurs in either of the input elements, values are automatically changed in the unordered list. The application produced by this code can be seen in Figure 7.1.	41

Chapter 1

Introduction

HotDrink [26], a JavaScript library for creating multi-way data-flow constraint systems is used throughout the thesis and the tools developed are built on top of it. In this thesis we create a visual editor for multi-way data-flow constraint systems. In the editor we can design form applications with GUI elements and create HotDrink constraint systems to handle the data-flow. The editor consists of three layers, illustrated in Figure 1.1.

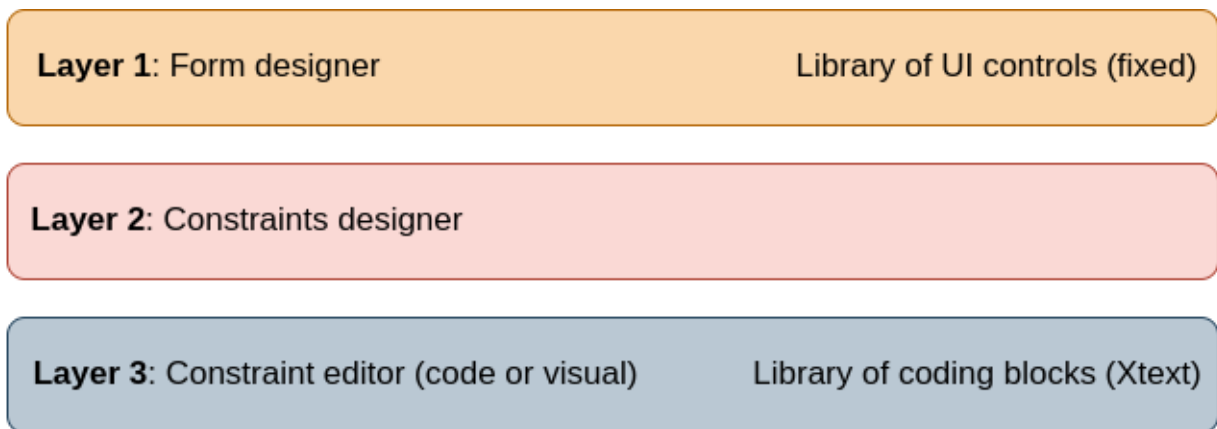


Figure 1.1: The 3-layer architecture of the visual editor.

The *form designer* (layer 1) is where the GUI elements are created and modified. The form designer contains a library of GUI elements, which the user can drag and drop onto the form. The *constraints designer* (layer 2) is where the user can create constraints and add methods to these constraints. The constraints are created in the same window as the form designer, where the constraint is connected to the GUI elements. The *constraint editor* (layer 3) consists of two editors: a *visual programming interface* and a code editor. The visual programming interface is where the user can edit methods visually. The

visual programming interface includes a library of coding blocks, which can be connected to the input and output nodes. The code editor contains code for each method, and the developer can edit methods using JavaScript code.

To make the visual programming interface more customizable, we created a domain specific language (DSL) for creating coding blocks to be used in the interface. This makes it possible for others to change the standard library and make their own coding blocks, instead of relying on the standard library. The DSL is implemented with Xtext [18] and it has a variety of different features, which makes it possible to create all kinds of coding blocks.

When the user is satisfied with the form application, the user can both test it out by running the application inside the visual editor and export the code to run in the browser.

We evaluated the visual editor by implementing GUIs with it, and assessing the outcome and the process. Concretely, we create a tax form that contains multiple data-flows, using HotDrink both both with and without our visual editor. The form provides a benchmark to compare the different methods of creating HotDrink applications. The prototype evaluation allowed us to evaluate if the visual editor can create the same applications as the non-visual editor and to evaluate the performance of the visual editor.

To test the usability of the visual editor we conducted a usability test on five users. They were asked to perform a series of tasks to create an application with the editor. The test was also conducted to get feedback on usability and to get a better understanding of how users interact with our visual editor.

1.1 Thesis outline

Chapter 1 An introduction to GUI programming using multi-way data-flow constraint systems and how we are using them to create visual diagrams.

Chapter 2 Background information about HotDrink and visual programming.

Chapter 3 Overview of the visual editor for HotDrink, including its features and limitations.

Chapter 4 Discussion about important parts of the implementation of the visual editor.

Chapter 5 A case study, implementing a Norwegian tax form using standard HotDrink with Hypertext Markup Language (HTML) and comparing it to implementing it using

our visual editor.

Chapter 6 A report of usability tests to test the usability of the editor, with results and feedback.

Chapter 7 Related work, looking at a number of different related works and comparing them to our implementation.

Chapter 8 A conclusion and discussion of some possible future work.

Chapter 2

Background

2.1 Motivation

There are many GUIs in the world and many of these are bad GUIs. Developers often make mistakes when developing GUIs and the users have to deal with the consequences; sometimes these mistakes are so bad that the users have to abandon the application. One such example is the GUI for a covid vaccination booking system, shown in Figure 2.1, where the user is not able to submit a valid input, even though it is valid according to the GUIs error message.

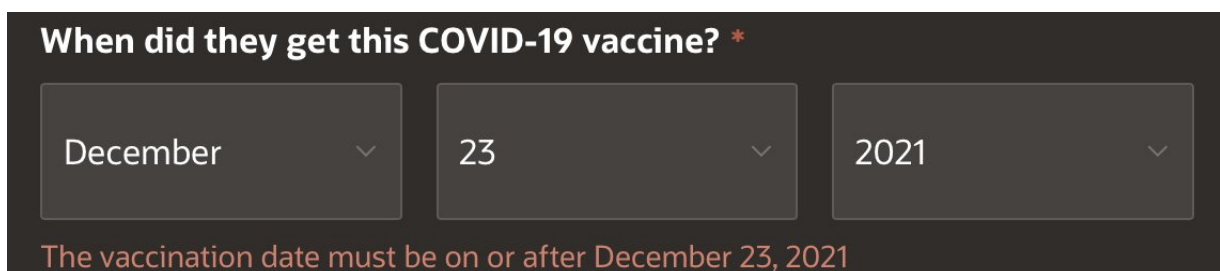


Figure 2.1: A covid vaccination form, where the user is not able to submit valid input.

Another example of a bad GUI is Figure 2.2 that shows a system for applying for Finnish citizenship. To apply, the user needs to enter all their travels abroad, but the user cannot remove the first trip, because of wrong validation: even though data is entered for the first trip, a validation error is shown saying that the data is missing.

In GUI programming there are many interconnections between elements, and these can be hard to program correctly and maintain. The above examples are consequences

Olen tehnyt matkoja ulkomaille tai oleskellut ulkomailta Suomeen muuttamisen jälkeen ?

Kyllä

Ei

Ilmoita alla matkasi:

Matkan kohde

Matkan kohde puuttuu!

Matkan tarkoitus

Matka alkoi

Matkan alkuaika puuttuu!

Matka päättyi

Matkan loppuaika puuttuu!

Matkan kohde

Matkan tarkoitus

×

Matka alkoi

Matka päättyi

Matkan kohde

Matkan tarkoitus

×

Matka alkoi

Matka päättyi

Matkan kohde

Matkan tarkoitus

×

Figure 2.2: A form for applying for Finnish citizenship, where the user is not able to remove the first trip. "Matkan kohde puuttuu" means that the destination of the trip is missing, "Matkan alkuaika puuttuu" means that the start time of the trip is missing and "Matkan loppuaika puuttuu" means that the end time of the trip is missing.

of this. GUI programming can be tedious; the code managing a GUIs event handling is often complicated, time-consuming to write and error prone. Studies [28, 33] have shown that the total development time of user interfaces is between 30 and 60 percent of that of the whole application.

A promising technology for less tedious GUI programming is constraint systems, which are a way to describe the relationships between elements. HotDrink is a GUI framework that uses constraint systems in order to create GUIs that are easy to use and maintain. While HotDrink tackles the problem of event handling code in many ways, there is currently no visual editor that makes it easy to create HotDrink applications with low code or no code.

This thesis focuses on making it easier for programmers and UX designers to visually understand the logic behind multi-way data-flow constraint systems by allowing them to create constraint systems using a visual editor. The visual editor has features that make it possible to create form-based applications without writing any code: a visual programming interface will create code blocks that generate code. The programmer can also combine visual specifications with regular JavaScript code.

While HotDrink is a programming framework directed to developers, we are interested in creating an interface for both non-developers and developers. With the visual editor, more users can be involved in the design of an application. As a result, more users might use the HotDrink library and understand the value of multi-way data-flow constraint systems.

HotDrink's current DSL for specifying constraint systems may feel unfamiliar compared to mainstream programming languages. A HotDrink specification essentially describes a graph, which may be difficult to see from a textual representation. A visual syntax is therefore more natural for constraint system specifications.

The goal of this thesis is to show that data-flow constraint system based GUI programming makes visual specification of GUIs feasible, and that implementing visual cues to users is straightforward in the chosen approach. GUIs should be easy to develop to keep time to market low and be cost competitive. A visual editor for HotDrink that saves development time is beneficial to the user.

2.2 Constraint system

In GUI programming constraint systems can be helpful in different programming tasks. A constraint system consists of variables and constraints, where constraints are the relations between the variables. A constraint system can automatically maintain a set of relations between variables, specified by constraints. In a constraint system, if the value of one variable changes so that one or more constraints are violated, a constraint solver can compute and assign new values for other variables so that the violated constraints are re-enforced [29].

A multi-way data-flow constraint system breaks down the relation that should hold for the constraint's variables to a set of functional dependencies. To specify a multi-way data-flow constraint, the programmer must thus implement a set of functions. It is the programmer's responsibility to guarantee that these functions always compute a result that satisfies the constraint [29].

2.3 HotDrink

HotDrink is a JavaScript library for developing web user interfaces [23]. It allows developers to create multi-way data-flow constraint systems. Constraint systems are specified either using HotDrink's DSL, which is designed specifically for creating constraint systems, or with JavaScript using HotDrink's Application Programming Interface (API) directly.

Listing 2.1: Temperature converter using HotDrink's DSL.

```
1 var celsius, fahrenheit;  
2  
3 constraint TemperatureConverter {  
4   toFahrenheit(celsius -> fahrenheit) => celsius * (9/5) + 32;  
5   toCelcius(fahrenheit -> celsius) => (fahrenheit - 32) * (5/9);  
6 }
```

The main concepts of a HotDrink constraint system are *components*, *constraints*, *methods* and *variables*. Listing 2.1 shows an example constraint system that could be used in a GUI that converts temperature values from `fahrenheit` to `celsius` and vice versa, written using HotDrink's DSL. This specification defines one HotDrink component, a container of constraints and variables. On the first line, we declare the component's variables, in this case `fahrenheit` and `celsius`. On line 3 we define a constraint with two methods, `toFahrenheit` and `toCelcius`. The first method converts the `celsius` variable to `fahrenheit` and the second method converts the `fahrenheit` variable to `celsius`. With this constraint in place, HotDrink will ensure that the two variables always represent the same temperature value, in different units. If we change the value of the `celsius` variable, the `fahrenheit` variable will automatically be updated according to the given formula, and vice versa.

Programs developed using HotDrink follow the Model-View-ViewModel (MVVM) pattern. In this pattern the model is responsible for the business data of the application. The view is responsible for presenting the data to the user and the viewmodel is responsible for supplying and managing the data and handling user actions. HotDrink's purpose is to implement the view-model part of MVVM [26].

To use HotDrink in a web application, *bindings* have to be added between HotDrink and the GUI elements. One-way data-flow from HotDrink to GUI elements is created by subscribing to changes on HotDrink variables, and updating the GUI when any changes occur, or subscribing to changes in the GUI and updating HotDrink variables when changes occur. Both directions of subscribing can be implemented between a GUI element

and HotDrink variable, which creates a two-way binding. With such bindings in place, the data in the GUI elements will stay in sync with the data in the constraint system. Since HotDrink variables are *observables* [13], it is easy to implement a suite of functions that create bindings between the view and the viewmodel.

2.4 Visual programming

A language that uses graphical elements to represent an application, where the graphical elements can be converted to code is called a visual programming language (VPL) [20].

Usually, programming happens by writing text in a code editor, where programmers have to learn concepts of programming to create applications and understand programming language syntax. For non-programmers, this is not ideal as users rather want to create applications with a process that makes sense to them [19]. Visual programming has the potential of making it easy to create simple applications using graphical elements, instead of worrying about programming language syntax. Another potential benefit of visual programming is that it requires less typing, which reduces the likelihood of typing errors. The visual notation may also be easier to get started with: instead of having to learn a possibly complex syntax of a new programming language, the programmer can focus on learning the concepts of the language.

A well-known example of a VPL is Blockly, a language developed by Google [2]. Blockly programmers use code blocks to visually create applications. Code blocks can only be composed in ways that form syntactically valid programs. Programmers do not thus have to worry about syntax. Blockly is widely used to introduce new learners to programming, to help them understand simple programming concepts. An example of a Blockly application is illustrated in Figure 2.3.

The visual presentation of Blockly emphasizes scopes and loops, which is a good fit with traditional programming. An alternative to block-based representation is a graph notation, with nodes representing computations and edges the flow of data. See Figure 2.4 for an example of such a visual programming diagram. This representation is a suitable method for programming with data-flow constraint systems.

2.5 Technologies

In building our programming environment, we relied on several tools and technologies. We give a brief overview of these tools below.

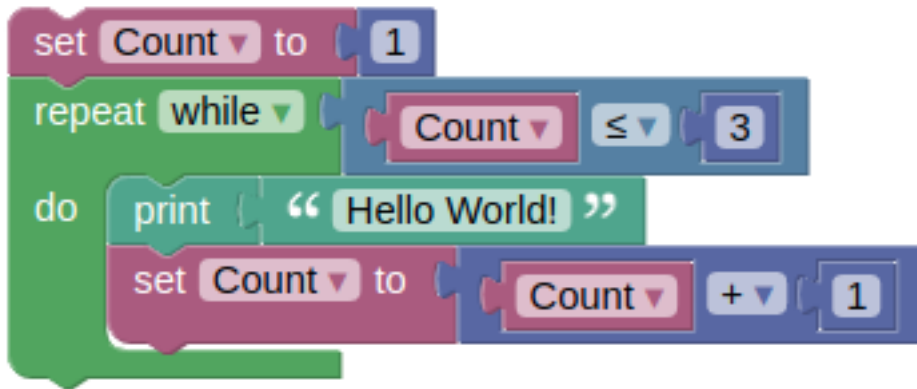


Figure 2.3: Example of a Blockly application, printing "Hello World!" three times [2].

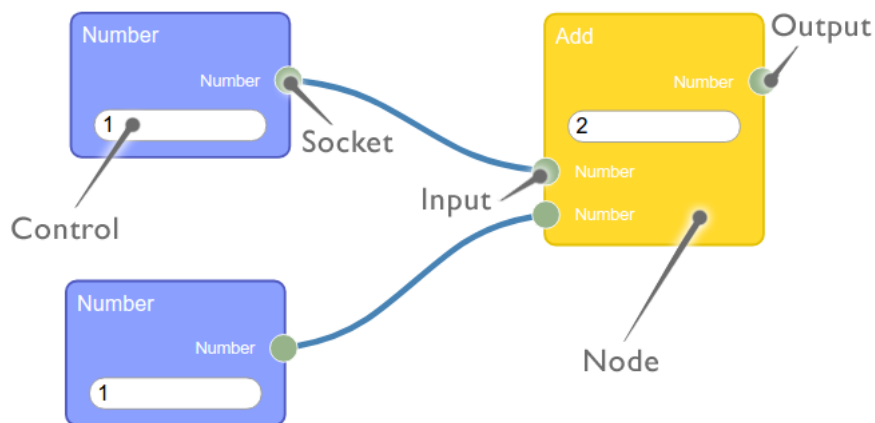


Figure 2.4: Example of a visual data-flow diagram, adding two numbers together [15].

2.5.1 TypeScript

TypeScript is a statically typed programming language that builds on JavaScript. It allows developers to write type-safe code and later convert TypeScript to JavaScript, which means that TypeScript runs anywhere JavaScript runs [6]. Because of its static typing TypeScript code is less error-prone than JavaScript code, and TypeScript code is considered to be easier to maintain. In this thesis, TypeScript is used to create the visual editor and it is used in combination with React.

2.5.2 React

React [14] is a JavaScript library for building user interfaces. React's goal is to make it easy to create interactive GUIs. React is component-based and it uses a declarative programming paradigm to build GUI components that are rendered to the DOM. React handles all state of a GUI: each component has its own state. React responds to all state changes by re-rendering the component. This architecture has shown to be suitable for creating web applications with different scales of complexity [14]. In the visual editor, state is handled by React and every GUI element is a React component. To get an idea of how React works, see Listing 2.2.

Listing 2.2: A counter example implemented in React with hooks [5]. In React, hooks can be created to define state in the application; here we use a hook to hold the counter value. Using `React.useState` we get a value and a function to set the value. We define our own increment function to increment the count, and call this when the user clicks the button. The incremented value will then be updated in the JSX tree (JSX is React's way of defining HTML and JS together).

```
1 export function Counter() {
2   const [count, setCount] = React.useState(0);
3
4   function increment() {
5     setCount(count + 1);
6   }
7
8   return (
9     <div>
10      <h1>Counter</h1>
11      <p>{count}</p>
12      <button onClick={increment}>
13        increment
14      </button>
15    </div>
16  );
17 }
```

2.5.3 Eclipse Xtext

Xtext is a framework for the development of programming languages and DSLs. From a grammar and semantics specification, Xtext produces a parser, linker, type checker and compiler, as well as editing support for any editor that supports the Language Server Protocol [18]. In this project, we use Xtext to create a DSL for programming blocks to a visual diagram, as explained in Section 3.2.2.

Chapter 3

A visual language for HotDrink specifications

The purpose of creating a visual editor for HotDrink is to allow users to visually create HotDrink applications with constraint systems. As discussed in Section 2.3, constraint systems consist of components, constraints, methods and variables. The editor needs to be able to create these and bind them to GUI elements.

The visual editor consists of two views: the form designer and the constraint editor. The form designer is the main view, where the user creates GUI elements and links them together with constraints. The constraint editor is where the user edits constraints. In the constraint editor, the user can choose between a visual editor and code editor.

Our tool consists of three layers:

- **Layer 1:** The *design editor* for adding and editing GUI elements.
- **Layer 2:** The *visual editor* for creating constraint systems and connecting them to graphical user interface elements.
- **Layer 3:** The *constraint editor*, which defines the logic of the constraint system.

3.1 Form designer

Layer 1 consists of three parts: a library of GUI elements, a canvas where the application is designed, and a properties panel where the user can edit the properties of a selected element or constraint.

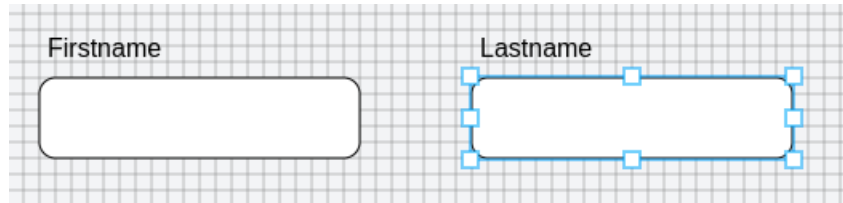


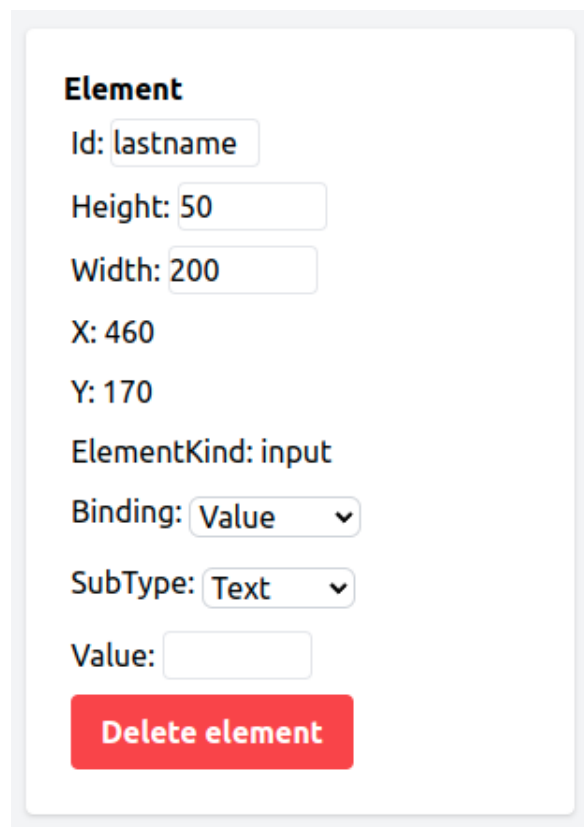
Figure 3.1: A form with fields `firstname` and `lastname`, with `lastname` selected (Layer 1).

The library of GUI elements shows thumbnails of full elements, mimicking what they will look like on the canvas. To add a new element, the user can drag and drop a miniature element onto the canvas. When dropping an element onto the canvas, it becomes a part of the user's application. Elements on the canvas can be selected and dragged around, to produce the desired layout of the application. In addition, users can drag the edges of each element to resize them to a desired width and height. Elements cannot be dragged or resized outside the canvas border. See Figure 3.1 for an example of a form application, where the `lastname` element is selected.

The *properties* panel shows the properties of the currently selected element. Figure 3.2 shows the properties of the selected `lastname` element from Figure 3.1. On the properties panel, the user can inspect and edit some of the properties of the element, such as *id*, *height*, *width*, *value*, *elementkind*, *subtype* and type of *Binding*. The *subtype* and type of *Binding* are dropdown menus with multiple options. The *subtype* property represents the type of the input element, such as text, number, date, button, etc. The type of binding represents what attribute the binding is going to bind to, for example, value, disabled or checked.

Constraints are created in the *new constraint*-mode, which the user can enter with the *Create Constraint* button. In this mode, every available element on the canvas will be highlighted (in green), and the user can click on an element to select it. The user can select any number of elements, and confirm the creation of a constraint after the desired elements are selected. The newly created constraint will then appear on the canvas next to the chosen elements as a circle, linked together with red connections. The connections will be either read connections, which means that the constraint can use the value of an element, or as a write connection, which means the constraint will be able to update an element's value. The constraint that is created is empty; to add logic to the constraint one needs to add methods.

Methods are created in the *new method*-mode, which the user can enter by selecting a constraint and clicking the *Add Method* button. Every element that is connected to



Element

Id: lastname

Height: 50

Width: 200

X: 460

Y: 170

ElementKind: input

Binding: Value

SubType: Text

Value:

Delete element

Figure 3.2: The properties panel shows the properties of the selected lastname element (Layer 1).



Figure 3.3: Highlighted elements while in *new constraint*-mode (Layer 2).

the selected constraint will then be highlighted (in green). The user can then select the output variables of the method, which can be one or many. By clicking the confirm button the visual editor will create a new method inside the constraint, and the method will be linked to the selected elements as output connections.

Similar to elements, the user can also drag constraints around. This is purely a matter of convenience for the developer: the position of a constraint on the canvas has no impact on the form design itself as constraints are not a visible part of the application at runtime.

As seen in Figure 3.1 and Figure 3.3, the canvas has a grid on the background. The grid has a snapping feature to make it easier to position elements. With the grid, users can see exactly where they have to place elements to get them in the correct x and y positions compared to other elements. If the user misses by a few pixels, the snapping feature will help to position the element correctly. The grid is not visible to the end-user of the form, it is only for designing the application.

3.1.1 Visual representation of constraints

Layer 2 is the constraint builder, where users can create multi-way data-flow constraint systems. Multi-way data-flow constraint systems are best visualized as a graph, as the underlying structure is an oriented bipartite graph [29]. Our visual editor visualizes the constraint system as graphs, our representation consists of two different graphs. The first one concerns the constraint view, where there are two types of nodes: constraint-nodes and variable-nodes with their connections being undirected. The other one is the method view, where there are two types of nodes: method-nodes which are subnodes of the constraint-nodes and variable-nodes with their connections being directed.

The methods of a constraint are visualized as enclosed in the node that represents the constraint. The methods are shown as a list; the user can click on each method to see and edit either the code of the method or its visual representation in a dialog box.

The visual representation of a constraint can have a number of different connections, and each connection can either be from the variable to the constraint or from the constraint to the variable. The former means that a method in the constraint will be able to *read* values from the variable, and the latter that a method in the constraint will be able to *write* values to the variable. Which kind of a connection the user needs to create depends on how the data-flow is supposed to be implemented according to the business logic of the application the user is creating. The data-flow can also be *bidirectional*, where a constraint can both read from and write to a particular variable.

In HotDrink, the set union of all input and output variables in each method must be the same for every method in a constraint. Our approach enforces this rule by automatically making all variables in a constraint at least input variables for all methods. The user can visually see which methods are connected to which variables with the arrows from the constraint. An arrow pointing to an variable is an output connection and a connection without an arrow is an input connection to the constraint.

For each constraint, the methods must adhere to a structural rule: for any two methods in a constraint, *m1* and *m2*, the set of output variables of *m1* cannot be a subset of the set of output variables of *m2*, and vice versa. This creates an upper bound of how many methods a particular constraint can have. For example, if a constraint has two connections to elements, say, to `firstname` and `lastname`, then the user can create a total of three different methods, one with `firstname`, one with `lastname` and one with both as output variables. However, the first of these cannot appear simultaneously with either of the latter ones, since the output variables of the latter are subsets of those of the former.

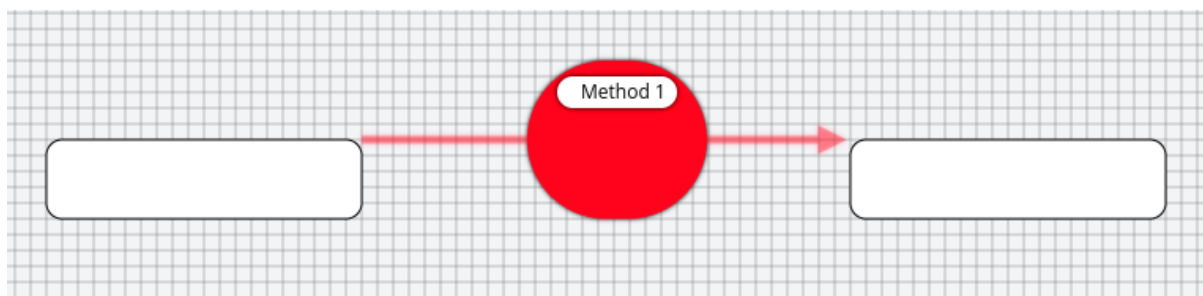


Figure 3.4: A constraint connected to two variables, with a method with one output variable (Layer 2).

In Figure 3.4, the constraint only has one output variable, and one method named `Method1` writing to this output variable. The method will be able to read values from both variables, but it can only write a value to the output variable. The output connection has an arrow pointing to the variable it writes to, whereas the input connection has no arrow indicator, but is merely a line.

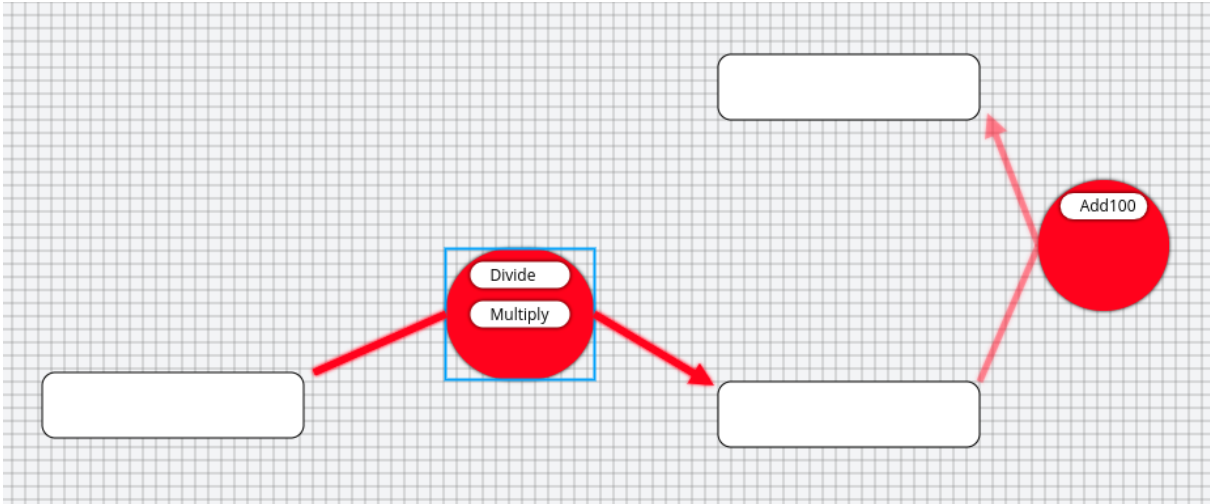


Figure 3.5: Hovering over a method will highlight the data-flow of the method. Here the effect of hovering over the method `Divide` is shown (Layer 2).

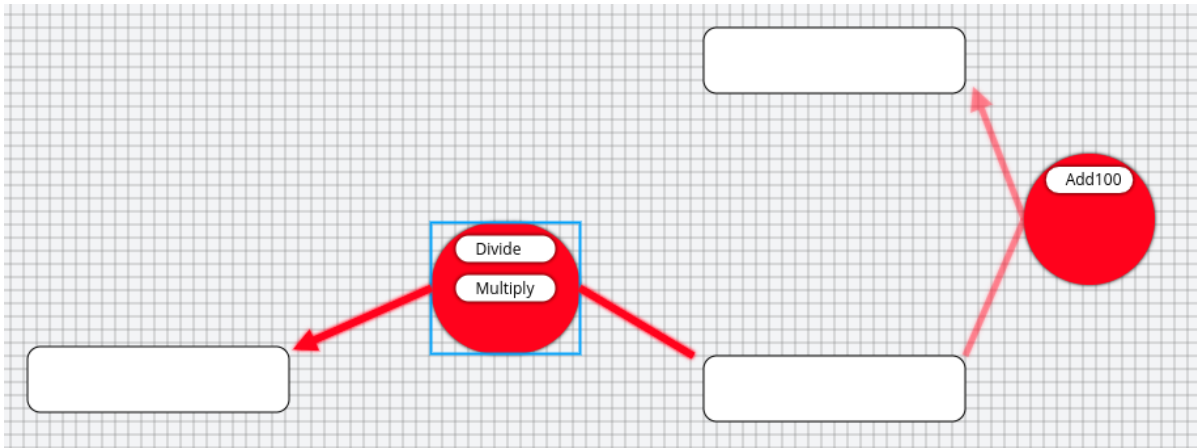


Figure 3.6: Hovering over a method will highlight the data-flow of the method. Here the effect of hovering over the method `Multiply` is shown (Layer 2).

Hovering over a method will highlight the read and write connections of the method. The parts of the data-flow that are not relevant for that specific method, such as other constraints and other method connections, will be grayed out. If a constraint has an arrow pointing to a variable, and the user hovers over a method that only reads from that variable, the arrow will be grayed out while the connection will be highlighted. Examples of both full arrows and partly grayed arrows are shown in Figure 3.5 and Figure 3.6.

Example: calculating the area and perimeter of a rectangle

Consider a constraint system example called WHAP, where W stands for width, H for height, A for area and P for perimeter of a rectangle. The constraints in the WHAP system ensure that the relations $A = wh$ and $P = 2(w+h)$ always hold. In HotDrink the code can be written as follows [25]:

Listing 3.1: WHAP implemented in HotDrink.

```
1 component whap {
2   var A=100, w, h, p;
3   constraint Pwh {
4     m1(w, h -> p) => 2*(w+h);
5     m2(p, w -> h) => p/2 - w;
6     m3(p, h -> w) => p/2 - h;
7   }
8   constraint Awh {
9     n1(w, h -> A) => w*h;
10    n2(A -> w, h) => [Math.sqrt(A), Math.sqrt(A)];
11  }
12 }
```

The component consists of two constraints. **Pwh** defines the relationship between the width, height and perimeter. **Awh** defines the relationship between the width, height and area. The constraint system can be visualized as shown in Figure 3.7.

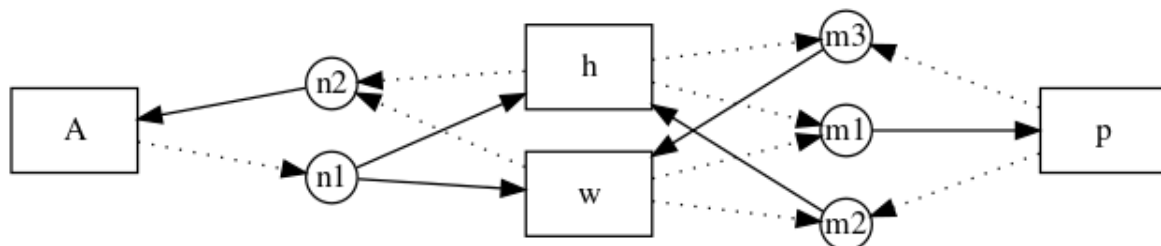


Figure 3.7: The WHAP constraint system example illustrated as a graph [25].

When figuring out how to visualize constraints, we took inspiration from the kind of graph representation of constraint systems shown in Figure 3.7. A WHAP constraint system can be implemented in our visual editor and visualized as in Figure 3.8. In an application developed for real users, the position of input boxes would be different, but here we chose a layout that matches the layout the WHAP constraint system in Figure 3.7.

Our visual editor implementation of the WHAP constraint system looks similar to the illustration Figure 3.7. The difference is that the visual editor groups the methods to make it easier to see which methods are connected to which constraints. Another difference is that the underlying graph has more connections, as in the visual editor every connected element to the constraint is an input variable for every method. In the WHAP

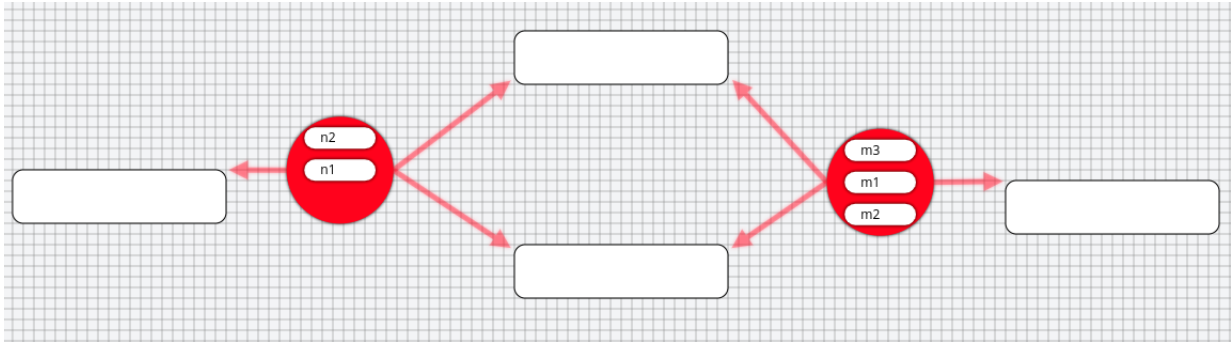


Figure 3.8: WHAP example visualized in the visual editor (Layer 2).

illustration the inputs are only added to the relevant methods. Each method is labeled the same as in the illustration, and every method has one output variable, except **n1** which has two. The constraint connections are best visualized using the hover feature to see the data-flow of each method.

3.1.2 Running and exporting

To test the application, the user can click on the *Run* button. This will open a dialog window showing what the application will look like. The application is fully working with an active constraint system where the constraint circles and grid from the form designer are hidden. When the user is ready to distribute the application, the user can click on the *Export* button. This will make a zip file, containing every asset needed to run the application in a browser, including the constraint system. Unpacking the zip-file and opening the HTML file will provide the user with a working application.

Figure 3.9: An example of *run-mode* with the implementation of WHAP.

3.2 Constraint editor

Layer 3 is the constraint editor, where users can edit the implementations of the methods of constraints, i.e., add logic to constraints. Adding logic can be done in two different ways: using the code editor or the visual editor.

3.2.1 Code view

The intended user group for the code view is software developers that know JavaScript. In the code view, the developers have full flexibility to write their logic without any restrictions, compared to the visual view where the user is restricted to visual code blocks added from the library of code blocks, as explained in Section 3.2.2.

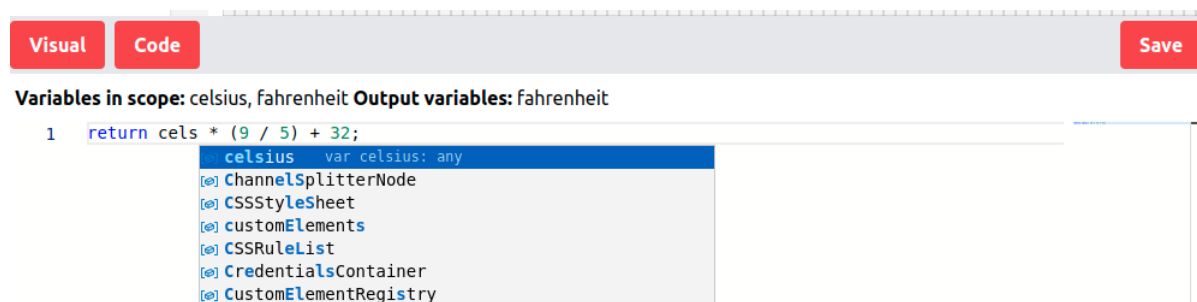


Figure 3.10: Codeview implementation of converting `celsius` to `fahrenheit`, showing autocomplete when writing `celcius`. (Layer 3).

The code view consists of a JavaScript code editor built with Monaco, the same editor that powers Visual Studio Code [9]. The editor features syntax highlighting and autocomplete for JavaScript, as shown in Figure 3.10. In this view, the user can write code for a specific method in the constraint system. The code has to return some value to be a valid method, without a return value the method will have no effect. To interact with the elements connected to the constraints, the method code needs to use the input variables. Input variables are provided in the editor for the method the user is editing. That way the user knows which variables are in scope. Figure 3.10 shows that `fahrenheit` and `celsius` are available when editing the method. Output variable(s) are also shown, and are most important when dealing with multiple outputs. When returning a value in a method with multiple outputs, the user returns an array with the values. The order of the values in this array has to be the same as the order of the output variables, otherwise the values will be written to the wrong output variables. When the method is complete, clicking the *Save* button will update the constraint system with the new method.

3.2.2 Visual programming interface

Compared to the code view, the visual programming interface is more restricted. The user can only add predefined blocks to the interface and connect them to the input and output variables of the method. The reason behind making the visual view restricted is to make it more intuitive for non-programmers. That way methods can be created by both programmers and non-programmers.

To make the interface consistent, the code blocks are added the same way as elements to the form builder. The user can drag and drop the block from a list of miniature blocks onto the interface and drag them around on a similar canvas to the form designer. Input and output variables are added to the interface automatically according to the specification of the method provided in the form designer. If two elements are connected with a constraint, with one input element and one output element, the visual programming interface will create one input and one output block, which can be connected with a logic block. Sockets are the connections of a block, which can either be input or output. To connect an input or output block with a logic block, the user has to click on the input and output socket on each block, and a connection will be formed. When a connection is formed from an input block to an output block, our application will generate code in the background for the method, with the specification of the logic block and add it to the constraint system when the user clicks the *Save* button.

Methods created with the visual programming interface can be changed in the code view if the user wants to expand or change the logic of the method. If the method is changed in the code view, it can not be converted back to the visual programming interface. Enabling this is planned as future work, for further details see Chapter 8.

Figure 3.11 shows an example method in our visual editor. The method has two inputs, `num1` and `num2`, both input variables of type number in the form designer. They are both connected to the logic block in the middle with connections between the sockets. The output of the logic block is the output result, which is connected to the output variable, which is also an input variable of type number. The red crosses on the top right corner of each block are for deleting the block; they are shown when editing the method. When this method is saved, it will generate code as in Listing 3.2. The code will find the maximum value of the two numbers and return it as the result.

Listing 3.2: Generated code output of maximum value example. The first line finds the maximum of the two input variables, and the second line returns the maximum value (Layer 3).

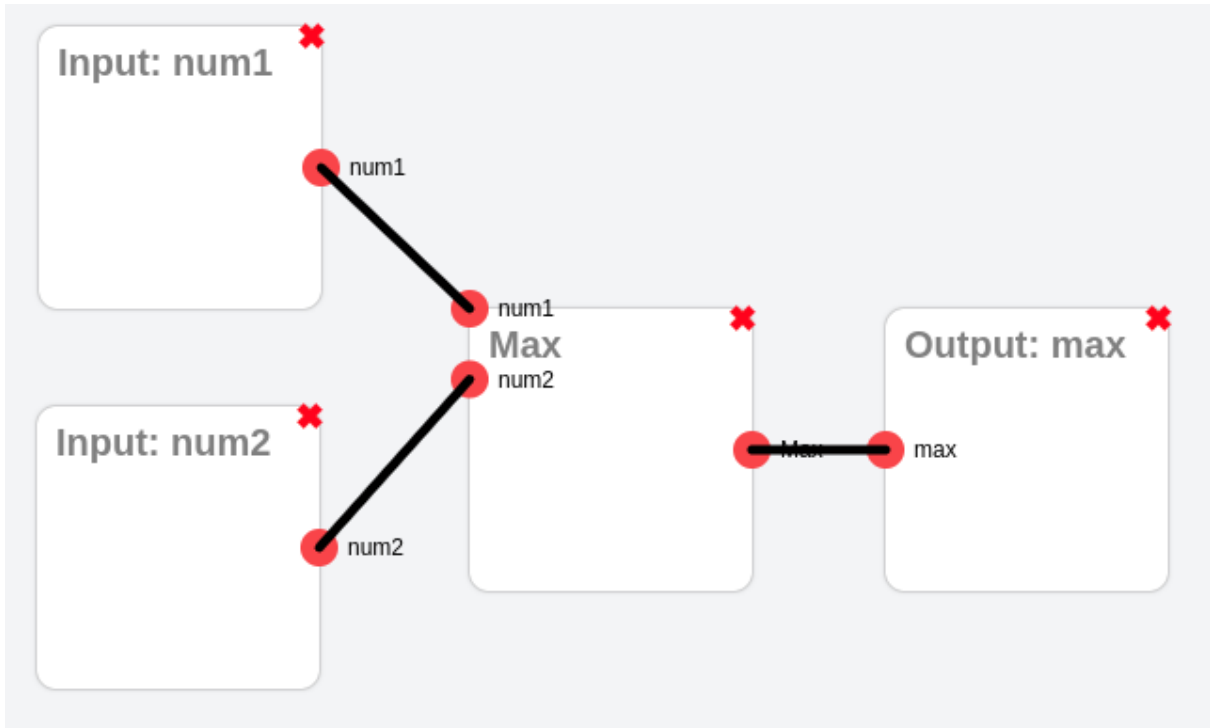


Figure 3.11: Visual programming interface calculating the max value of two inputs (Layer 3).

```

1 const Max = Math.max(num1, num2)
2 return Max;

```

A DSL for visual blocks

To make it easier for non-programmers to use the visual programming interface, we created a standard library of blocks, and a DSL for extending this library with new blocks. The DSL was implemented using Xtext. The DSL is a language that is used to create visual programming blocks, in text form, that can be imported into the visual programming interface. The goal of the DSL is to make it easy to make a library of blocks that can be used to create simple HotDrink applications. These blocks should not be complicated and should be easy to use. They need to contain at least one input and one output since they are designed to use the input value in the code and output a different value. The DSL ensures that these blocks are created in a type-safe way and with a structure that can be imported to our visual programming interface.

We have created a standard library for the visual programming interface, containing common logic blocks that are often used. Table A.1 shows the standard library. The

DSL language can also be used to create custom blocks, to be imported to the visual programming interface. This makes our application extensible since the user can create their own blocks and import them to the visual programming interface.

Listing 3.3: The grammar of the DSL in EBNF (Layer 3).

```
1 blockDeclaration = "block" name "{" { inputDeclaration } |  
    ↪ paramDeclaration | [ codeDeclaration ] "}" ;  
2 codeDeclaration = "code" code ;  
3 inputDeclaration = "input" name ;  
4 paramDeclaration = "param" name ":" paramType ;  
5 name = "a".."z" | "A".."Z" | "0".."9" ;  
6 code = ''' ... ''' ;  
7 paramType = "number" | "text" | "date" ;
```

The grammar of the DSL is shown in Listing 3.3. The DSL's main entity is the **block**, which is used to define a new visual block. A block can contain any number of **inputs**, a **param** and one **code**. The **inputs** are used to define the input variables to the block, which are meant to be connected from within another block. The **code** is used to define the code logic of the block. The code written in the code field is the code that will be the body of a HotDrink method. It must be written in JavaScript. As we can see in Listing 3.4, the JavaScript code is written within triple quotation marks. The **param** is used to define the type of the input field, which is placed on top of the block in the visual editor and can be used together with the **input** variables in the code.

Listing 3.4: Example of a visual block implemented using the DSL (Layer 3).

```
1 block Add {  
2     input number1  
3     input number2  
4     code '''number1 + number2'''  
5 }
```

Listing 3.4 shows an example of an addition block defined using the DSL. The block has two inputs: **number1** and **number2**. The purpose of this block is to return the sum of both input numbers.

Listing 3.5: Contains example of a visual block using the DSL (Layer 3).

```
1 block Contains {  
2     input str  
3     param textBox: text  
4     code '''str.contains(textBox)'''  
5 }
```

Listing 3.5 shows an example of a block that checks if a string contains a certain text provided in a text box. The block has one input, **str** which is the string to be checked. The code of the block returns a boolean value, which is true if the string contains the text, and false if it does not.

All the blocks that are available in the standard visual block library are listed in Appendix A.

Chapter 4

Implementation

This chapter explains implementation details of different parts of our application. The discussion is not exhaustive, we focus on those features described in Chapter 3 that are interesting, tricky, or otherwise worth further elaboration.

4.1 Form Designer

Our implementation of the form designer is built using a library called *Konva*, which is a JavaScript library for drawing complex canvas graphics [30]. We utilize Konva to draw the elements onto the canvas and visualize the constraints. Konva supports events such as drag-and-drop, hover, and click events. Konva draws graphics pixel by pixel, which makes it possible to have full flexibility of what is drawn on the canvas. With Konva, developers can add elements to the canvas with specific x and y coordinates, and design the application exactly as their users want.

Before we adopted Konva, we considered implementing the form designer as an HTML editor, where the user could add HTML elements to the canvas and manipulate them. When dragging and dropping elements onto the page, the elements could be added to the right place in the HTML tree and visualized on the canvas. This approach would also have supported events, as events are built into HTML with JavaScript.

We did not go for this approach for a few reasons. HTML elements could be drawn using standard HTML, but other elements, such as constraints, would have to be drawn using SVG, Cascading Style Sheets (CSS), or HTML Canvas which are harder than using

the drawing library Konva. Another reason why we did not go for this idea was that we wanted the application to be as generic as possible so that not only HTML could be used in the future, but other formats or frameworks as well. Instead of the GUI being stored as HTML, we thus created our own format to represent its elements, with additional properties. This format is connected with Konvas API for drawing graphics, which gave us full control over the data and the graphics in the form designer. This format is exportable to HTML, and can be exported to other formats as well by implementing a converter between the format and our data type.

In HotDrink the user can output multiple values from each method. Therefore, in the code view the user can output multiple values using a list of variables. The length of the list of variables should be the same as the number of outputs from the method. Multiple outputs are not, however, supported by the visual programming interface. The reason is that we wanted the visual programming interface to be a simple diagram with some number of input fields, one logic block and only one output block. Multiple output blocks would have complicated the logic blocks and required more outputs to be connected from the logic blocks. Keeping the visual diagram simple makes the visual editor more welcoming to non-programmers. In addition, the standard library only consists of blocks with one output block, and one logic block. This means that the user can only connect one output block to one logic block. There are no cases in the library that need multiple outputs, but if in the future such need arises, multiple outputs could be added to the interface as well.

4.2 Visual programming interface

As discussed in Chapter 2, there are different ways of doing visual programming. Our main focus was to make this interface easy and understandable to everyone. We looked at a number of different visual programming approaches and concluded that a diagram-like syntax was the best fit for our application. Another approach we considered was using Blockly, but we decided against it because using Blockly requires its users to understand programming concepts, at least on some level [31].

Instead of Blockly, we looked into a JavaScript visual programming framework called Rete. Rete is a modular framework that allowed us to create a node-based visual editor [15]. We believe that this approach is easier for end users to understand and use than Blockly. In Rete, one can create multiple nodes that can be connected with connections

between input and output sockets. These nodes can be connected as a graph, and the graph can generate JavaScript code line by line from each node.

We did not end up using Rete for the visual editor. We found that Rete both had a lot of bugs and was outdated and not actively maintained. Therefore, we decided to make our own visual programming interface, inspired by the Rete framework. Our implementation had roughly the same features and it ended up having a lot fewer problems and thus it is better for the user to use.

Using Rete one could create complicated graphs and have a lot of flexibility in the way that one connects nodes. To keep method representations simple, we constrained the visual editor to only contain one or more input nodes, one main logic node and one output node. This way all programs will look quite similar, and the user will not have to learn complex programming concepts to use the editor or to connect many different logic blocks to create a program. This works reasonably well with HotDrink, since methods in HotDrink applications are usually not that complicated, at least when we are creating simple form applications.

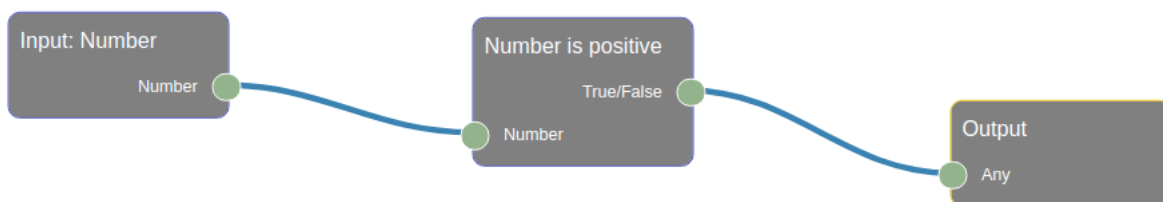


Figure 4.1: An example method implementation using our visual programming interface built with Rete. This method checks if a number is positive and returns true or false accordingly.

Code generation was an interesting task, as visual programs are not mimicking exactly what a textual program should do line by line. One of the main tasks was to share variables between blocks. Each block has its own scope, and variables are not shared by default. We implemented a way to share variables between blocks, by letting each block pass its variables to the next blocks. The input and output sockets were used to keep

track of which blocks the variables should be passed to. This way the next block in the diagram can use the variable name from the previous block to get the value.

An example of the generated code is shown in Listing 3.2. Here we pass two variables from the two input nodes to the max block. The max value is then calculated in this block and generates a variable with the result. The max value is passed to the output node, which returns the passed value.

DSL for code blocks

The DSL for creating blocks for the library has to be interpreted by the visual editor. Since the code written in the DSL is not a format that is understood by the visual editor, we made the code convert to a readable format. Since JavaScript Object Notation (JSON) is a format that is both easy to write and understand, we decided to use JSON. Using Xtext, each time we make a change in a DSL program, the program is automatically converted to JSON, which is then exported from the Integrated Development Environment (IDE) and processed by the visual editor. This allows others to use and import programs written in our DSL, as for example if someone wants to create a different library for the visual editor.

4.3 HotDrink

4.3.1 Components and constraints

In the background we use HotDrink's API to create components and constraints. Our convention is that there is only one component for each form application. This component includes all variables (GUI elements) and all constraints that are connected to the GUI elements. In addition to creating components and constraints, we also create the binders between the HotDrink variables and GUI elements when the application is run and exported to make the HotDrink constraint system connect to the GUI elements.

4.3.2 Serialization and deserialization

To export a constraint system, it has to be serialized into a storable format. There was no implemented functionality to do this in the HotDrink framework. Therefore we decided to implement serialization and deserialization as part of the framework.

The storable format has to store the state of the constraint system, which means everything that is needed for a constraint system to be reconstructed. This includes the components, constraints, methods and variables. This state from the storable format is then deserialized into a new constraint system when an application is loaded. The format is a JSON file, which is a format that is easy to both to write and parse, and understand.

The biggest challenge of storing constraint systems in a JSON format is that HotDrink contains circular references for variables, and JSON does not support this. Therefore, a simple `toJson` method in JavaScript is not enough in this situation. We needed to implement custom serialization and deserialization to handle circular references. With this storable format, we can export a constraint system from the visual editor, and use it in an application. For example, the temperature converter in Listing 2.1 would be exported as a JSON file as shown in Listing 4.1.

Listing 4.1: Storable format of temperature converter example in JSON.

```

1 {
2   "components": [
3     {
4       "name": "Component1",
5       "variables": [
6         {
7           "name": "celcius"
8         },
9         {
10          "name": "fahrenheit"
11        }
12      ],
13      "constraints": [
14        {
15          "name": "C2",
16          "constraintSpec": {
17            "methods": [
18              {
19                "nvars": 2,
20                "ins": [
21                  0,
22                  1
23                ],
24                "outs": [
25                  1
26                ],
27                "code": "(celcius, fahrenheit) => {\n
                ↪          return celcius * (9/5) +
                ↪          32\n
                ↪        }",
28                "promiseMask": [
29                  0,
30                  0
31                ]
32              },
33              {
34                "nvars": 2,
35                "ins": [
36                  0,
37                  1
38                ],
39                "outs": [
40                  0
41                ],
42                "code": "(celcius, fahrenheit) => {\n
                ↪          return (fahrenheit - 32)
                ↪          * (5/9)\n
                ↪        }",
43                "promiseMask": [
44                  0,
45                  0
46                ]
47              }
48            ]
49          },
50          "optional": false
51        }
52      ]
53    }
54  ]
55 }

```

Chapter 5

Case study — Norwegian tax form

To evaluate the programming experience that our visual editor offers, we created an example application in both standard HTML with JavaScript and HotDrink, and in our visual editor. The goal of this case study was to evaluate how the development process of HotDrink applications is affected by the visual specification and compare the results of different methods of creating HotDrink applications. We were also interested to see if we can create the same application in our visual editor to see if there are limitations that the visual editor imposes.

Yearly income	<input type="text" value="700000"/>	Tax percentage	<input type="text" value="30"/>
Months spent in Norway in tax year	<input type="text" value="10"/>	Finnmark deduction	<input checked="" type="checkbox"/>
Deduction	<input type="text" value="111630"/>		
Tax	<input type="text" value="98370"/>	Net income	<input type="text" value="601630"/>

Figure 5.1: Norwegian tax form using HotDrink, JavaScript and HTML.

The application we are evaluating is a Norwegian tax form, where the user can fill out parts of the form and let HotDrink calculate the other fields using different formulas. When changing fields, other fields will be automatically updated according to a formula

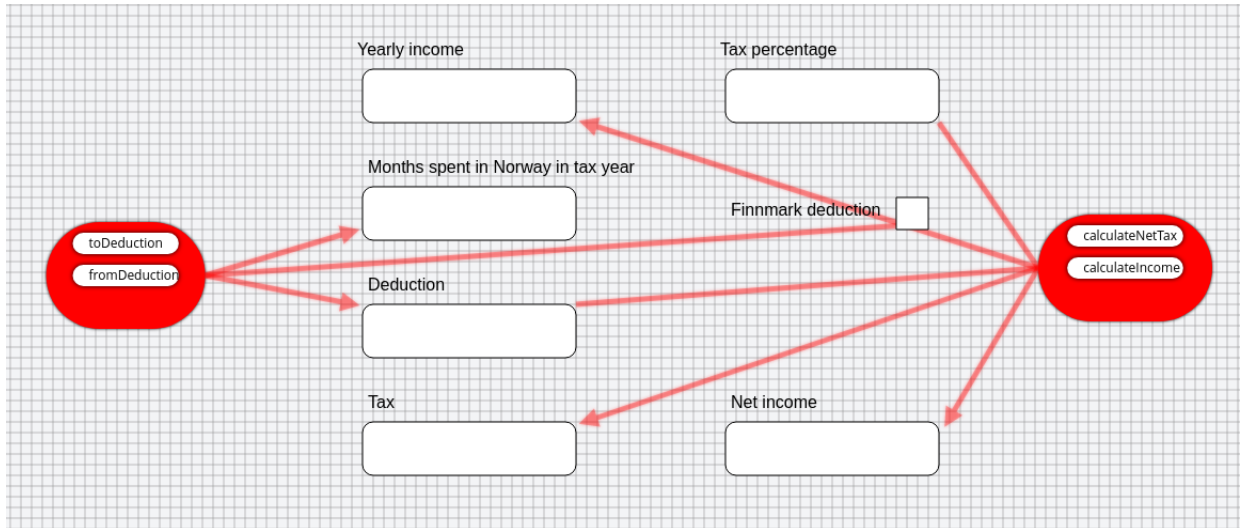


Figure 5.2: Norwegian tax form in the visual editor's design mode.

in one of the methods in the constraint system. The form includes a number of data-flow constraints which makes it a perfect example to showcase HotDrink's usage. The purpose of the tax form is to calculate the tax amount and net income from a given income, tax percentage, and deduction. Alternatively, the application lets the user calculate the net income when either the tax or the net income changes.

A prototype of the tax form can be seen implemented in the visual editor in Figure 5.2.

Listing 5.1: Norwegian tax form constraints in HotDrink, calculating tax and income, with deduction.

```
1 var income, percentage, time, finnmark = false, deduction, tax,
  ↪ net_income;
2
3 constraint {
4   (income, percentage, deduction -> tax, net_income) => {
5     var newTax = (income * percentage / 100) - deduction;
6     var newNet_income = income - newTax;
7     return [newTax, newNet_income];
8   }
9   (tax, net_income, deduction, percentage -> income) => {
10    var newIncome = parseInt(net_income) + parseInt(tax);
11    return newIncome;
12  }
13 }
14
15 constraint {
16   (finnmark, time -> deduction) => {
17     var timeDeduction = 9163 * time;
18     var finnmarkDeduction = 20000;
19     if (finnmark) {
20       return timeDeduction + finnmarkDeduction;
21     } else {
22       return timeDeduction;
23     }
24   }
25   (deduction, finnmark -> time) => {
26     if (finnmark) {
27       return (deduction - 20000) / 9163;
28     } else {
29       return deduction / 9163;
30     }
31   }
32 }
```

The example has two constraints, which handle all logic of the application. The first constraint on line 3 in Listing 5.1 has two methods for calculating to and from income and net income, with tax and percentage. The first method calculates the tax and net income from the given income and percentage and returns both of these variables. That means that both of these will be updated in the GUI if the constraint is enforced using the first method. The second method calculates the income from the given tax and net income. This is the inverse of the first method and is used to calculate the income when either the tax or the net income changes.

The second constraint has two methods, one calculating the deduction from the number of months in Norway and if the user is eligible for the Finnmark deduction. The amount of months spent in Norway is multiplied by 9163 NOK to get the total amount of month deduction. Finnmark deduction is a fixed amount of 20000 NOK. The deduction is added to the total amount of month deduction. The deduction is then returned as the total of Norwegian kroner to be deducted from income. The other method calculates how many months the user has spent in Norway from the total amount of deduction, and takes into account whether the user is eligible for the Finnmark deduction.

Developing the tax form in the visual editor is markedly different from programming it by textually writing code. In the latter, the programmer has to track a lot of details, variable names, syntax and how the dependencies are formed in the code, where, as in the visual editor, the programmer can see the direct mapping of the dependencies and see exactly which fields are dependent on each other. With a single glance, the programmer can be reassured that the network of dependencies is as expected. When writing code textually, these details are not visible to the programmer, and the programmer has to develop them themselves. As a result a visual editor has the potential to give a better, faster and less error prone programming experience than a textual code editor.

The tax form we have developed is only one example, but it is indicative, as it already makes the developing experience of HotDrink application much more pleasant. During this thesis work we did not have the time and resources to conduct a larger study, but in the future we believe it would be beneficial to conduct a study where different developers would implement a larger number of GUIs, for example other tax forms, and compare the results.

Chapter 6

Usability test

6.1 Setup

We conducted a usability test on the behavior of the visual editor, where users were asked to create a form using our tool and test it out. The test was conducted using the users' web browsers. Following commonly accepted advice [34] we selected five users as subjects: with adding more users, we are not likely to learn that much new about our application's usability.

The users were asked to follow the steps below to create a width, height, and area calculator of a square.

1. Add three input fields to the canvas.
2. Set the id of these input fields to `height`, `width` and `area`.
3. Change these input fields to be of *subType* `number`.
4. Create a constraint between the three input fields.
5. Create two methods within the constraint, one for calculating `area` from `width` and `height` and one to calculate from `area` to `width` and `height`.
6. Add logic to the method that calculates the area using the visual programming interface, the method should return *width * height*.

7. Write JavaScript code for the method with output variables `width` and `height`. The method should return the square root of `area` for both outputs: \sqrt{area} . In HotDrink to return multiple variables, the user has to return an array with these variables.
8. Run the program and test with different numbers.

If the user was a non-programmer, they were be asked to skip step seven, as this is a programming task.

6.2 Results

We conducted the test on a total of five users. We chose users with different backgrounds. Three of them had either never used a visual programming tool, or had barely used a visual programming tool. Two of them were quite familiar with other visual programming tools. Four of these were programmers, and one was a non-programmer. The non-programmer was asked to skip the programming task.

The five users used a range of different operating systems and some different browsers. Three of them used Mac OS, one used Windows, and one used Linux. Three of them used Chrome, one used Firefox, and one used Safari. As this is a web application, it is important to test with different devices and browsers to ensure that the application works the same on all of them.

Table 6.1: A table of the results of the usability test. Green (●) means that the participant completed the task successfully. Yellow (◐) means that the participant completed it, but it either took a long time or multiple tries. Red (◑) means that the participant failed to complete the task and we had to intervene to help the participant complete the task. White (-) means that the task was skipped.

	Step							
Participant	1	2	3	4	5	6	7	8
Participant 1	●	●	●	●	◐	●	●	●
Participant 2	●	●	●	●	●	●	◐	●
Participant 3	●	●	●	●	◐	●	●	●
Participant 4	●	●	●	◑	◑	◐	-	●
Participant 5	●	●	●	●	●	◐	●	●

Four of the five users completed the task successfully, one failed to complete some of the steps and we had to intervene to help them complete the task. All of these participants had no experience with this exact tool prior to the test, which means that some yellow fields were expected. The most important part of the task is that they were able to complete it, and the result shows that most were able to manage just that.

6.2.1 Feedback

After the usability test, we got a good idea of what limitations the visual editor had. We have identified some things that should be improved to make the visual editor more usable. Below we list current limitations that should be improved, based on what the users did and what they said.

Layer 1 — Form designer

In the form designer, dragging-and-dropping elements onto the canvas was a bit different on Mac OS compared to the other operating systems. On Mac OS, the elements that were dropped floated back to the start before they were shown on the canvas. The other operating systems did not have this problem, and the elements were dropped on the canvas without floating back.

Layer 2 — Creation of constraints

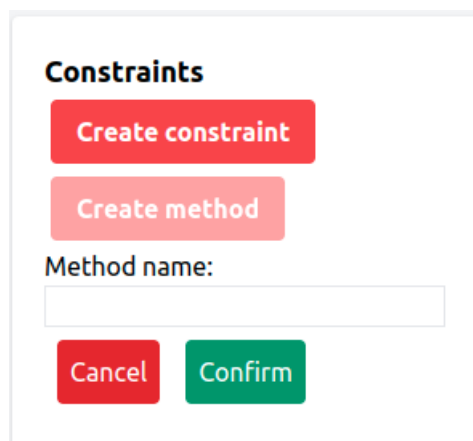


Figure 6.1: When clicking on *Create method*, an input field and two buttons appear.

When creating methods on a constraint, an input field for the name with a *confirm* and *cancel* button is displayed, as shown in Figure 6.1. When this *Create method*-mode is active, users have to click on the elements that should be output variables and write a name for them. Some of the participants thought the *Confirm* and *Cancel* buttons were only for the name of the method, but the *Confirm* button is also for creating the method. Users do get a message that tells them that they also need to click on the elements on the canvas to create the method, but some of the users did not read it. Therefore, it might be a good idea to make this message more clear, for example by presenting it in an alert box. The users do get an error message if they try to create a method without selecting elements, so they finished the task without help either way.

One participant ended up creating methods with wrong output variables, and this participant realized this after the method was created and looked for an undo button. We do not have an undo and redo option in the visual editor so the participant had to delete the method instead. Deleting is our intended solution to this problem, but an *undo* button might be a better solution.

Layer 3 — Constraint editor



Figure 6.2: The top part of the constraint editor.

One of the main problems we encountered while doing the usability tests was that the participants did not understand how to get out of the constraint editor. There is a *Save* button in the right corner of the editor, but this was not intuitive enough. *Save and exit* might be a better name for the button, to make it more clear that this is the way to exit the constraint editor.

A suggestion from one of the participants was to include the name of the method that is being edited at the top of the constraint editor. This could be added at the top of the editor between the buttons, to make it easier for the users to understand what method they are editing.

In the visual programming interface, the user needs to click on the sockets to create a connection. This was intuitive for most of the participants, but some of them tried to

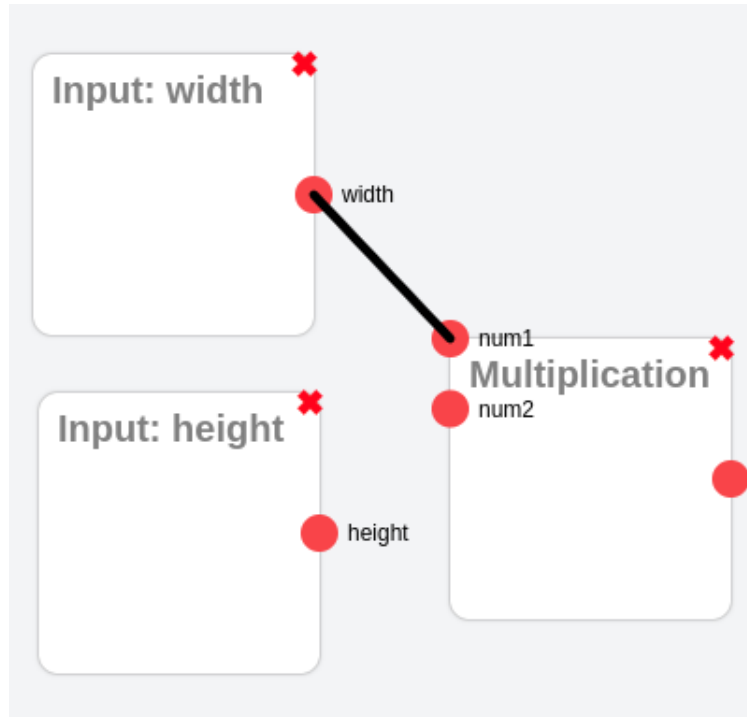


Figure 6.3: Visual programming blocks showing sockets between *width*, *height* and a multiplication block.

drag from them first. This might be a better solution since some of the participants were slightly confused when they could not drag from the sockets.

Finally, we got feedback that the visual programming interface was small in size compared to the content provided inside. The visual programming blocks take up a lot of space, and there is not a lot of space inside the visual programming interface. It might be a good idea to make the visual programming interface bigger so that the user can see more of the content.

6.3 Summary

The feedback we got is a good indicator of what future work should be done. Many minor things could be done to improve the usability of the visual editor. It would be beneficial to have a new usability test after these improvements and see how the results improve. Overall, we got a good idea of what the limitations of the visual editor were.

Chapter 7

Related work

There are a number of different works related to our application, which we will discuss in the following sections.

7.1 Constraint systems

Besides GUI programming, constraint systems are used in other areas of software engineering too. We discuss some of the most prominent applications below.

7.1.1 Amulet Environment

The *Amulet Environment* [32] is a programming environment that incorporates a number of design and implementation innovations, such as new models for objects, animation, output, input, commands, but most importantly, constraints. Amulet stands for *Automatic Manufacture of Usable and Learnable Editors and Toolkits*. It was implemented in C++, and it ran on a wide range of operating systems. The Amulet environment integrates constraint solving with an object system. The object system is a prototype-instance based system, which is used to represent the objects in the constraint system. In the Amulet environment, a user operates with *slots*, which can contain either values directly or expressions that when evaluated will emit a value. Like HotDrink, when there is a change in the value of a slot that is referenced by a constraint, the corresponding value expression is automatically re-evaluated [32].

7.1.2 ConstraintJS

ConstraintJS is a JavaScript library for creating constraints in dynamic web applications. ConstraintJS enables the developer to create constraints, which are relationships that are declared once and then automatically maintained. These constraints can be used to control content and control display [36]. Similar to HotDrink, ConstraintJS allows developers to specify their own bindings between constraint variables and Document Object Model (DOM) elements [35]. ConstraintJS allows the developer to use mostly HTML and CSS to declare the interactive behavior, rather than using large amount of JavaScript [36]. Unlike HotDrink, ConstraintJS does not provide support for multi-way constraints, but rather only supports one-way constraints, as the authors see multi-way constraints as an unnecessary complication. instead of a benefit [36]. As seen in the WHAP example, using multi-way data-flow constraints is beneficial; implementing the example manually would require more work and be more error prone.

Listing 7.1: A ConstraintJS example, listing commander names in a list [35]. The following is a template that is connected to the ConstraintJS constraint system. The DOM elements are bound to the constraint system, such that when a change occurs in either of the input elements, values are automatically changed in the unordered list. The application produced by this code can be seen in Figure 7.1.

```
1 <label>Title:</label>
2 <input class='form-control' type='text' cjs-out='demonym'
3     placeholder='Demonym' value='Commander' />
4
5 <label>Members (comma-separated):</label>
6 <input class='form-control' type='text' cjs-out='items'
7     placeholder='Comma-separated items'
8     value='Kirk, Spock, Sulu, Uhura' />
9
10 <hr/> <label>Result:</label>
11
12 <ul class="list-group">
13     {{#each items.split(",")}}
14     <li class="list-group-item">
15         <strong>{{demonym}} {{@index}}</strong>: {{this}}
16     </li>
17     {{/each}}
18 </ul>
```

7.2 Form builders

Form builders are often used to create forms by dragging-and-dropping widgets on a canvas. The following work uses form builders similar to our visual editor.

Title:

Members (comma-separated):

Result:

Commander 0: Kirk
Commander 1: Spock
Commander 2: Sulu
Commander 3: Uhura

Figure 7.1: Example of comma separated list of names, using ConstraintJS. The code for this application is shown in Listing 7.1.

7.2.1 JotForm

JotForm is an online form builder with a range of different library components, such as input field, date-picker, dropdown list, checkbox, etc. These components can be dragged and dropped onto the canvas of the end-user application being developed. JotForm is mainly used to collect data from users and store the results in a database. Users can define conditional logic between components to create dynamic forms. The conditional creator supports most of the same operators as the form builder developed in this thesis. However, JotForm does not support presenting conditional logic in a diagram view, which is in contrast to our solution, which supports visualizing the logic and data-flow of the form. The target user base of JotForms is mostly non-programmers, who use the tool mostly for creating different types of surveys [4].

7.2.2 Microsoft Visual C# Express

Microsoft Visual C# Express is a lightweight IDE used to develop Windows desktop applications. The IDE contains an interactive GUI with a design mode that is used to visually create Windows Forms applications. The design mode includes a library of

components (such as buttons and text boxes), which are common GUI elements in such applications. Similarly to this thesis, the design mode also includes a properties panel, which lets the user control the properties of each component. For a more fine-tuned customization of the end-user application, a user can open the *code view*. The code view shows the code that is generated by the design mode; this is where the user can edit components as code, or to change the behavior of the application. The code view does not have the same visual features as the design mode to create components, but the code view is more flexible. The code is generated and edited in C#, which is the programming language used by the IDE. As Microsoft Visual C# Express is designed to be used not only by developers but also by hobbyists and students, a combination of both design mode and code view makes development suitable for a wider audience of developers [19].

Microsoft Visual C# Express achieves much of the same as our form builder, with similar features such as a design mode and a code editor. Their IDE does not however have a visual programming interface where users can program application behavior using visual components.

7.2.3 Delphi

Delphi is an IDE for creating native applications. The IDE has been used for a long time primarily for creating native Windows applications, but Delphi can now also compile natively to Android, iOS, macOS, and Linux [3]. Delphi is similar to Microsoft Visual C# Express in that it contains an interactive design mode with a code editor that works practically the same. Instead of C#, Delphi applications are written using a version of the programming language Pascal. The editor has been around for many years; the first version was released in 1995 and was called Borland Delphi [24]. The IDE has been expanding its functionality over the years, but Delphi's primary focus has always been on the visual programming environment for rapid creation of native applications.

7.3 Lowcode / no code environments

To allow non-programmers to develop applications, there exists a range of environments that do not require programming.

7.3.1 Pure Data

Pure Data can be used to create programs without writing a single line of code. Pure Data is a visual programming language that can be used to process and generate sound, video, 2D/3D graphics, interface sensors, input devices, and Musical Instrument Digital Interface (MIDI) [12]. In pure data, user draws logic and data-flow interactively with preprogrammed objects, which can be placed on a canvas. Data-flow is created by making visual connections between objects. Each of these objects performs a predefined task which can range from basic mathematical operators to advanced video encoding. Pure Data is directed at a number of different use cases, and can be utilized by performers, researchers, artists, and developers [12]. These users can use Pure Data for various domain-specific purposes, with a range of different library objects.

7.4 Visual programming environments

Similar to our visual programming interface, there are other solutions that aim to provide a visual programming environment for non-programmers. There is both evidence for and against visual programming; we provide a summary of the two.

7.4.1 LabVIEW

Laboratory Virtual Instrument Engineering Workbench (LabVIEW) is a visual programming environment designed for scientists and engineers, and is used to develop programs using a graphical notation. LabVIEW uses a powerful graphical programming language called *G*, which aims at a significant increase in productivity, when compared to textual programming [27].

Similar to our application, LabVIEW has a *front panel*, which is comparable to our *design view*. In the front panel, a user can drag components onto a canvas, drag them around, and resize them. The front panel includes a *run mode*, which is used to test the program in a live environment.

LabVIEW also contains a *block diagram*, which is a graphical programming interface for creating programs. The block diagram is similar to our visual language in the way that it has blocks that are connected together with wires. The data-flow of programs is built

using these wires with a range of different blocks from a block library. LabVIEW's block diagram tool is of course quite advanced, with many more features than our visual editor, such as the ability to create loops, functions, and provide error messages. LabVIEW is a tool aimed for professional programmers.

7.4.2 Empirical evidence for and against visual programming

Visual Programming Languages and the Empirical Evidence For and Against [39] paper by John Whitley discusses the evidence for and against the use of VPLs. The paper summarizes empirical data relevant to VPLs and the results of research on VPLs [39].

Visual programming in the real world [22] is an industry-based study that found empirical evidence for visual programming. The study claims that there are productive use cases for the use of VPLs in real-world programming tasks [39]. In this study the researchers had two teams of programmers, one team was using LabVIEW, a VPL, and the other team was using text-based programming. Both teams were given the task of developing an application for a customer in three months. Both teams received the same amount of funding and the same requirements. After three months, the LabVIEW team had a better result than the text-based team. The LabVIEW team was able to complete all of the requirements given by the customer, including extra features. The text-based team did not finish all of the original requirements and needed more time to finish [22].

One of the reasons behind LabVIEW's success was that the team had closer contact with the customer, the customer was able to understand more of the development process as they used a VPL. While the text-based team had not met their customers before the presentation after three months [22]. If the text-based team also had met their customer, the result might have had a slightly different outcome, but since the LabVIEW team was that much more efficient, closer contact with the customer would likely not have made a huge difference for the outcome.

Whitley found that in real-world programming tasks, visuals outperformed text notation in either time or correctness and even in both. Whitley also found that in situations where people designed or worked with problem-solving, they performed better when information was presented in a consistent and organized matter, like with a VPL [39].

Historically flowcharts have been the primary visual tool for programmers. Two studies, one by Shiderman [38] and one by Ramsey, Atwood, and Von Doren [37] studied the

use of flowcharts as an aid to textual code, but they did not find any evidence that they were a benefit to use with textual code [39]. Whitley found that the use of VPLs depends on the task that is to be performed. It is important to consider the effects of VPL on a range of different tasks. Another important issue to consider is whether the effect of VPLs is large enough to be of practical interest [39]. An example of this is the spreadsheet, a widely used tool for data analysis, and it is clear that it is a useful application.

There is both empirical evidence for and against VPLs. The empirical evidence for VPLs shows that there is a need for VPLs to be used in real-world programming tasks. The empirical evidence against VPLs shows that not every task is suitable for VPLs, and that the use of VPLs depends on the task that is to be performed. We argue that our VPL is useful in a number of different tasks, but some tasks are not suitable for our VPL. This concides well with Whitley's research.

7.5 Language workbenches

DSLs can be created with mainstream programming languages, but this often requires a lot of work. To minimize work and make the creation of DSLs easier, there are multiple Language Workbenches that help with the task.

7.5.1 Eclipse Xtext

As mentioned in Section 2.5.3, Xtext is a framework for developing programming languages and DSLs. The framework is used in the Eclipse IDE as a plugin. With Xtext, the user can build full-featured text editors for both general-purpose languages and DSLs. Building a grammar is easily done using the Xtext grammar language. The grammar language is similar to the Extended Backus-Naur Form (EBNF) [18]. When building a language, the user gets a fully-fledged IDE with language support for the new language, with syntax highlighting and code completion, out of the box.

7.5.2 Langium

Langium is an open-source language engineering tool, with support for the Language Server Protocol [7]. Langium is written in TypeScript and runs in Node.js. The goal

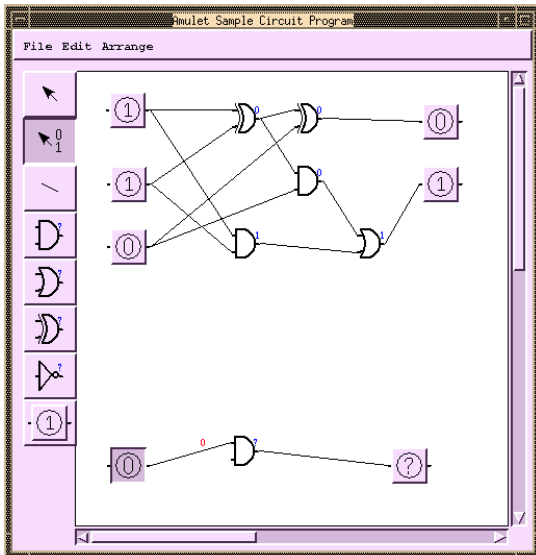
of Langium is to "bring language engineering to the next level", enabling the creation of domain-specific languages in VS Code, Eclipse and other IDEs. Langium is based on Xtext, and it has some of the same features, but with a different modern approach. Langium keeps the concepts that have made Xtext successful but lifts them onto another platform. The tool provides a grammar language for easily defining syntax rules and structure for languages [7].

7.5.3 Whole platform

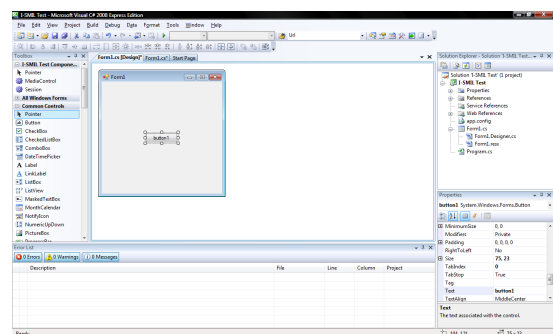
Whole platform is another Language Workbench. Like Xtext it is built on top of Eclipse. The workbench can create new languages and manipulate them using domain notations. With this workbench, domain experts can work together with programming experts, where the domain experts write the business knowledge, and the programming experts write the generators for the language [17].

7.5.4 MetaEdit+

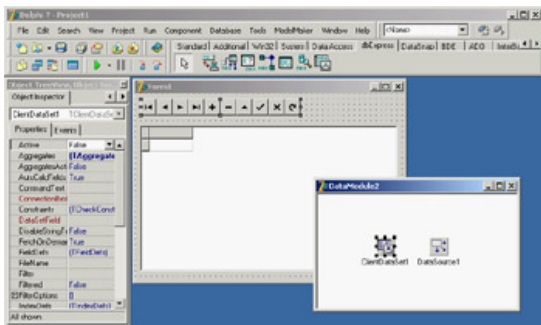
MetaEdit+ is a graphical language workbench for designing domain specific languages. MetaEdit+ includes a modeler, which provides a graphical interface through which to use the language created in the workbench. The primary focus of this language workbench is to create a tool that can be used to create domain specific languages without a single line of code [8].



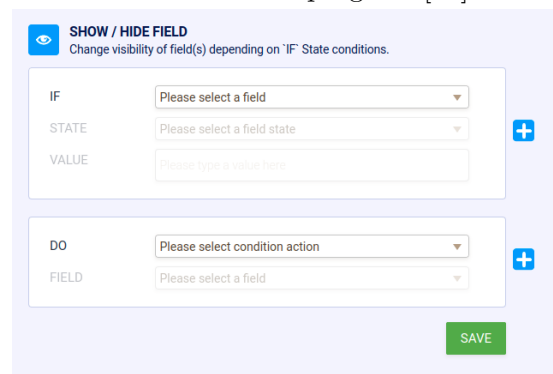
(a) A sample circuit, using the visual programming interface of the Amulet environment [32].



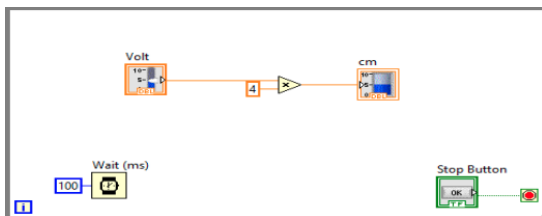
(b) The design mode of Microsoft Visual C# express, showing an example form, dragging elements from the library to the form will add it to the program [16].



(c) A form and a data module in the Delphi 7 IDE [11].

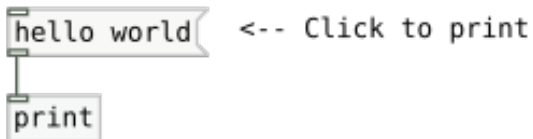


(d) The conditional creator of JotForm, showing how to add logic to JotForm [4].

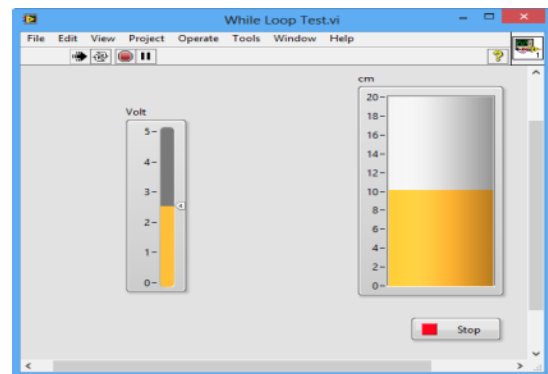


(e) Showing the volt to cm in LabVIEW block mode, where the logic of the application is developed [1].

"Hello World" in Pd!



(g) An example program in Pure Data that will output Hello world [21].



(f) Showing a comparison between volt and cm, with a stop button to end the program, in the LabVIEW design mode [1].

Figure 7.3: Some screenshots of the related works presented in this chapter.

Chapter 8

Conclusion and future work

8.1 Conclusion

In this thesis work we constructed a visual programming tool that can be used to create and visualize HotDrink applications. The thesis shows that a visual diagram helps the programmer develop and track the dependencies of user interfaces that use data-flow constraint systems. There is definitely audience for this program, it is suitable for every programmer and non-programmer that wants to create applications with dependencies between widgets.

Here we studied the complex dependencies that one encounters even within user interfaces of relatively simple everyday applications. Such dependencies have not been easily visualized in a visual representation before, since the dependencies tend to be implemented in a complicated and adhoc way using event handling. This work relies on data-flow constraint systems, enabling visual programming of these dependencies, and performs an early evaluation of the benefits of such tools. There is obviously a benefit of programming constraint systems visually versus textually as such tools provide the programmer with substantial aid. Our end goal is to integrate tools developed here as a mainstream tool for graphical interface programming.

8.2 Future work

Currently, we export the applications created in our visual editor to HTML and JavaScript. In principle, we could also export to other formats or even frameworks,

for example to React, Angular, or Vue.js. With multiple export targets, developers could choose which language or framework they want to continue with if developers want to further expand the application after using our visual editor.

Another possible expansion would be to include different graphic designs in the export. In React and other frameworks, some libraries can be used to style the elements in the application. We could use a component library from these frameworks and use it to style the elements in the application, for example, React's MaterialUI [10].

Another natural future extension is to support more types of elements in the visual editor's design mode. Currently, we support text, button, date, checkbox and number. We could add more types of elements, for example, image, audio, list, etc. This would add more use cases: users would be able to visually create HotDrink applications with more advanced elements.

There are also other possibilities regarding the visual programming interface. We made a standard library that is easy to use and is not very complicated. We could add a more advanced library that could cope with more complex operations, to make it more adaptable for professional users. Also instead of having support for only one logic node at a time, we could further expand this to support multiple logic nodes at a time. This would make it possible to chain operations together, for example first multiplying variables and then doing another operation on the result of the multiplication. Our reason for not doing this was to keep the interface simple and easy to use.

In our DSL the block's input and output variables are used as the names of the sockets in the visual programming interface. These are to describe what goes in and what goes out with each block. Since some of these blocks are very similar and have the potential to be reused, an idea worth exploring is to add inheritance of blocks. An example of reuse would be the `Addition` block and `Multiplication` block. These have similar inputs that could be inherited from an abstract block. That way we only would need to write the output code for each block.

Currently in the visual programming interface, code can be generated from visual blocks and edited in the code editor. This is not possible the other way around, code from the code editor cannot be edited in the visual programming interface. Such a round trip feature would be useful, but quite demanding to implement.

After the usability test, we got some insights from users about the usability of our visual editor. There are several smaller features that we could add or improve, mentioned in Section 6.2.1.

To further evaluate and test the visual editor, we could create larger applications with the editor, to experiment how the visual editor can be used in larger projects. Applications with many dependencies between widgets are of particular interest. As our visual editor is a working prototype, it can be used in real world projects to let users create applications using HotDrink.

List of Acronyms

API Application Programming Interface.

CSS Cascading Style Sheets.

DOM Document Object Model.

DSL domain specific language.

EBNF Extended Backus-Naur Form.

GUI graphical user interface.

HTML Hypertext Markup Language.

IDE Integrated Development Environment.

JSON JavaScript Object Notation.

LabVIEW Laboratory Virtual Instrument Engineering Workbench.

MIDI Musical Instrument Digital Interface.

MVVM Model-View-ViewModel.

VPL visual programming language.

Bibliography

- [1] Basic LabVIEW Programming.
URL: https://www.halvorsen.blog/documents/teaching/courses/labview_automation/labview_basic.php. [Accessed on *2022-03-21*].
- [2] Blockly.
URL: <https://developers.google.com/blockly?hl=nb>. [Accessed on *2021-12-02*]. A JavaScript library for building visual programming editors.
- [3] Delphi: IDE Software Overview — Embarcadero.
URL: <https://www.embarcadero.com/products/delphi>. [Accessed on *2022-01-20*].
- [4] Free Online Form Builder & Form Creator | Jotform.
URL: <https://www.jotform.com/>. [Accessed on *2022-02-03*].
- [5] Introducing Hooks — React.
URL: <https://reactjs.org/docs/hooks-intro.html>. [Accessed on *2022-04-11*]. A JavaScript library for building user interfaces.
- [6] JavaScript With Syntax For Types..
URL: <https://www.typescriptlang.org/>. [Accessed on *2022-01-15*].
- [7] Langium.
URL: <https://langium.org>. [Accessed on *2022-03-07*].
- [8] MetaCase — MetaEdit+ Workbench.
URL: <https://www.metacase.com/mwb/>. [Accessed on *2022-05-02*].
- [9] Monaco Editor.
URL: <https://microsoft.github.io/monaco-editor/>. [Accessed on *2022-01-25*].
- [10] MUI: The React component library you always wanted.
URL: <https://mui.com/>. [Accessed on *2022-05-09*].

- [11] An overview of the IDE: Delphi.
URL: <http://etutorials.org/Programming/mastering+delphi+7/Part+I+Foundations/Chapter+1+Delphi+7+and+Its+IDE/An+Overview+of+the+IDE/>. [Accessed on *2022-03-21*].
- [12] Pure Data — Pd Community Site.
URL: <https://puredata.info/>. [Accessed on *2022-01-07*].
- [13] Quickly understand JavaScript observables. Implement the Observable class from scratch and understand what makes observables different from promises..
URL: <https://www.stackchief.com/tutorials/JavaScript%20observables%20in%20%20Minutes>. [Accessed on *2022-04-08*].
- [14] React — A JavaScript library for building user interfaces.
URL: <https://reactjs.org/>. [Accessed on *2022-01-15*]. A JavaScript library for building user interfaces.
- [15] Rete.js.
URL: <https://rete.js.org/#/>. [Accessed on *2022-02-02*].
- [16] Visual Studio Express - A-SMIL.org.
URL: https://www.a-smil.org/index.php/Visual_Studio_Express. [Accessed on *2022-03-21*].
- [17] Whole Platform - Home Page.
URL: <https://whole.sourceforge.io/>. [Accessed on *2022-04-29*].
- [18] Xtext — Language Engineering Made Easy!
URL: <https://www.eclipse.org/Xtext/>. [Accessed on *2022-01-15*].
- [19] What is Visual Studio Express (VSE)? - Definition from Techopedia. November 2012.
URL: <http://www.techopedia.com/definition/24334/visual-studio-express--vse>. [Accessed on *2022-03-10*].
- [20] Introduction to Visual Programming Language. September 2021.
URL: <https://www.geeksforgeeks.org/introduction-to-visual-programming-language/>. [Accessed on *2021-12-02*].
- [21] Pure Data 2022.
URL: https://en.wikipedia.org/w/index.php?title=Pure_Data&oldid=1072958158. [Accessed on *2022-03-21*]. Page Version ID: 1072958158.

- [22] Ed Baroth and Chris Hartsough. Visual programming in the real world. In *Visual object-oriented programming: concepts and environments* pages 21–42. Manning Publications Co. USA January 1995. ISBN 9780131723979.
- [23] John Freeman, Jaakko Järvi, and Gabriel Foust. HotDrink: a library for web user interfaces. *ACM SIGPLAN Notices*48(3): 80–83. September 2012. ISSN 0362-1340. doi: 10.1145/2480361.2371413.
URL: <https://doi.org/10.1145/2480361.2371413>. [Accessed on 2021-11-10].
- [24] Zarko Gajic. The History of Delphi. March 2017.
URL: <https://www.thoughtco.com/history-of-delphi-1056847>. [Accessed on 2022-03-10].
- [25] Jaakko Jaarvi. Hotdrink documentation.
URL: <https://git.app.uib.no/Jaakko.Jarvi/hd4/-/blob/master/docs/tutorial/tutorial.org>. [Accessed on 2022-02-15].
- [26] Jaakko Jaarvi. Introduction to HotDrink.
URL: <http://hotdrink.github.io/hotdrink/howto/intro.html>. [Accessed on 2022-01-13].
- [27] Jim Kring Jeffrey Travis. *Labview for Everyone: Graphical Programming Made Easy and Fun*. Prentice Hall.2006. ISBN 0131856723; 9780131856721.
- [28] 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, USA2008. Association for Computing Machinery. ISBN 9781605582672. doi: 10.1145/1449913.1449927.
URL: <https://doi.org/10.1145/1449913.1449927>.
- [29] Jaakko Järvi, Gabriel Foust, and Magne Haverdaen. Specializing planners for hierarchical multi-way dataflow constraint systems. *ACM SIGPLAN Notices*50(3): 1–10. September 2014. ISSN 0362-1340. doi: 10.1145/2775053.2658762.
URL: <https://doi.org/10.1145/2775053.2658762>. [Accessed on 2021-12-02].
- [30] Anton Lavrenov. Getting started with react and canvas via Konva.November 2020.
URL: <https://konvajs.org/docs/react/index.html>. [Accessed on 2022-02-02].
- [31] Siti Nor Hafizah Mohamad, Ahmed Patel, Rodziah Latih, Qais Qassim, Liu Na, and Yiqi Tew. Block-based programming approach: challenges and benefits. In

Proceedings of the 2011 International Conference on Electrical Engineering and Informatics pages 1–52011. doi: 10.1109/ICEEI.2011.6021507.

- [32] Brad A. Myers. The Amulet User Interface Development Environment.
URL: <https://www.cs.cmu.edu/afs/cs/project/amulet/www/papers/videoabs.html>.
[Accessed on 2022-03-07].
- [33] Brad A. Myers and Mary Beth Rosson. Survey on user interface programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '92. pages 195–202. New York, NY, USA June 1992. Association for Computing Machinery. ISBN 9780897915137. doi: 10.1145/142750.142789.
URL: <https://doi.org/10.1145/142750.142789>.
- [34] Jakob Nielsen. Why You Only Need to Test with 5 Users. March 2000.
URL: <https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/>.
[Accessed on 2022-04-25].
- [35] Stephen Oney. ConstraintJS.
URL: <https://soney.github.io/constraintjs/>. [Accessed on 2022-03-01].
- [36] Stephen Oney, Brad Myers, and Joel Brandt. *ConstraintJS: Programming Interactive Behaviors for the Web by Integrating Constraints and States* pages 229–238. Association for Computing Machinery New York, NY, USA 2012. ISBN 9781450315807.
URL: <https://doi.org/10.1145/2380116.2380146>.
- [37] H. Rudy Ramsey, Michael E. Atwood, and James R. Van Doren. Flowcharts versus program design languages: an experimental comparison. *Communications of the ACM* 26(6):445–449 June 1983. ISSN 0001-0782. doi: 10.1145/358141.358149.
URL: <https://doi.org/10.1145/358141.358149>. [Accessed on 2022-01-31].
- [38] Ben Shneiderman, Richard Mayer, Don McKay, and Peter Heller. Experimental investigations of the utility of detailed flowcharts in programming. *Communications of the ACM* 20(6):373–381 June 1977. ISSN 0001-0782. doi: 10.1145/359605.359610.
URL: <https://doi.org/10.1145/359605.359610>. [Accessed on 2022-01-31].
- [39] K. N. Whitley. Visual programming languages and the empirical evidence for and against. 1996.

Appendix A

DSL standard library

Table A.1: Visual blocks available in the standard DSL library (Layer 3).

Block	Inputs	Params	Explanation
Add	2	0	Adds the two inputs together.
AddWith	1	1	Adds the input with number given in param.
Subtract	2	0	Subtracts the first input with the second input.
SubtractWith	1	1	Subtracts the input with the number given in param.
Multiply	2	0	Multiplies the two inputs.
MultiplyWith	1	1	Multiplies the input with the number given in param.
Divide	2	0	Divides the input dividend with the second input divisor.
DivideWith	1	1	Divides the input with the number given in param.
Mod	2	0	Modulo of the input dividend with the second input divisor.
ModWith	1	1	Modulo of the input with the number given in param.
LessThan	2	0	Returns true if first input is less than second input.
MoreThan	2	0	Returns true if first input is more than second input.
LessOrEqual	2	0	Returns true if first input is less or equal than second input.

MoreOrEqual	2	0	Returns true if first input is more or equal than second input.
IsPositive	1	0	Returns true if input is positive number.
IsNegative	1	0	Returns true if input is negative number.
IsZero	1	0	Returns true if number is zero.
IsOdd	1	0	Returns true if number is odd.
IsEven	1	0	Returns true if number is even.
Min	2	0	Returns the minimum number of the two inputs.
Max	2	0	Returns the maximum number of the two inputs.
Length	1	0	Returns the length of the input string.
Concat	1	1	Concats the input with the param.
Contains	1	1	Returns true if the input contains the param.
ToLowerCase	1	0	Returns the input string in lower case.
ToUpperCase	1	0	Returns the input string in upper case.
IsEmpty	1	0	Returns true if string is empty.
And	2	0	Returns true if both inputs are true.
Or	2	0	Returns true if one of the inputs are true.
Not	1	0	Returns the opposite of the boolean value in input.
IsTrue	1	0	Returns true if input is true.
IsFalse	1	0	Returns true if input is false.
IsBefore	1	1	Returns true if input date is before param date.
IsAfter	1	1	Returns true if input date is after param date.