University of Bergen
Department of Informatics

Tool support for specifying multi-way dataflow constraint systems

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

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

November, 2022

Abstract

Programming Graphical User Interfaces (GUIs) is a complex task. One reason for this is the way GUIs use complex dependencies between each other. This is where the constraint system library HotDrink comes in: HotDrink is a JavaScript library that allows the programmer to define complex constraints between GUI elements in a declarative way. This makes it possible to write complex GUIs in a simple way.

This thesis is motivated by the fact that HotDrink is a very powerful library, but it is not used by many developers. We believe that this is because HotDrink is missing dedicated tool support. This thesis presents tools that we have developed to support the HotDrink library; these tools are: *HDCode* and *HDDebug*. HDCode is an Integrated Developer Environment that supports HotDrink with syntax highlighting, validation, code completion, code generation and visualization capabilities. HDDebug is a extension for Google Chrome that lets debugging HotDrink code running on a webpage.

Acknowledgements

First and foremost, I would like to thank my supervisors Mikhail Barash and Jaakko Järvi for their support and excellent guidance. I would also like to thank Knut Anders Stokke for his expertise and development capabilities. Finally, I would like to thank my family and friends for their support and encouragement.

Mathias Skallerud Jacobsen
Monday 21st November, 2022

Contents

1 Introduction **1**

2 Background **3**

2.1 Constraint systems . . . 3

2.1.1 HotDrink . . . 11

2.2 Architecture of Google Chrome extensions . . . 15

2.3 Technology stack . . . 15

3 HDCode and HDDebug **18**

3.1 HDCode . . . 19

3.1.1 Support for the HotDrink DSL . . . 20

3.1.2 JavaScript code for the HotDrink API . . . 21

3.1.3 Demo application . . . 22

3.1.4 Visualizing HotDrink specifications . . . 24

3.1.5 HDCode in action . . . 25

3.2 HDDebug . . . 31

4 Implementation **33**

4.1 Integrated Development Environment . . . 33

4.1.1 Grammar . . . 34

4.1.2 Scoping . . . 34

4.1.3 Validation and manipulation . . . 35

4.1.4 Visualization . . . 38

4.1.5 Code generation . . . 39

4.2 Chrome extension . . . 40

5 User study **43**

5.1 Results . . . 45

5.1.1 Task 1 . . . 45

5.1.2 Task 2 . . . 46

5.1.3 Tasks 3 and 4 46
5.1.4 Task 5 47
5.1.5 Tasks 6 and 7 47
5.2 Summary 47

6 Related work 50
6.1 Integrated development environments 50
6.1.1 Visual Studio Code 51
6.1.2 Atom 51
6.1.3 PyCharm 51
6.1.4 Eclipse IDE 52
6.1.5 AWS Cloud9 52
6.2 Support for web framework in integrated development environments . . . 52
6.2.1 React in WebStorm 52
6.2.2 Vue.js in VScode 54
6.2.3 Ktor in IntelliJ IDEA Ultimate 55
6.2.4 Vaadin in IntelliJ IDEA Ultimate 55
6.3 Tools for debugging web applications 55
6.3.1 React Developer Tools 56
6.3.2 Redux DevTools 56
6.3.3 Angular DevTools 56
6.4 Language workbenches 57
6.4.1 Xtext 57
6.4.2 Spoofax 58
6.4.3 Rascal 58
6.5 Visual specification 58
6.5.1 LabVIEW 58
6.5.2 Graphviz 59

7 Discussion and future work 61

List of Acronyms and Abbreviations 63

Bibliography 64

A Grammar for the HotDrink DSL used in the implementation of the VSCode extension 69

B Binder functions generated by the demo application feature 72

C Coe validations supported by HDCode **73**

List of Figures

2.1 A graph showing the relations between the variables in the electrical bill example. Nodes represent variables and edges represent relations between variables. The Graphviz code for this graph, rendered by Graphviz, is generated by our tool based on the constraint system specification by our VSCode extension. . . . 7
2.2 An example of an undirected bipartite graph. . . . 9
2.3 Example of a constraint graph. . . . 10
2.4 Example of a solution graph, non-greyed out element are chosen as the solution graph. . . . 10
2.5 Plan for the constraint system in Listing 2.2. The non-grayed out vertices and edges are the solution graph. . . . 13

3.1 Auto-completion of a component in HDCode. . . . 20
3.2 Validation of a HotDrink DSL specification in HDCode. The developer is using variables that are not defined, which is an error, hence, the underlined syntax is in red. . . . 21
3.3 Generation of HotDrink API code from a HotDrink DSL specification. . . 23
3.4 Graph view of a HotDrink DSL specification that converts between the temperatures in degrees Celsius and Fahrenheit. . . . 25
3.5 Variable relations visualized as a graph. . . . 26
3.6 Initiation of variables using a quick fix 28
3.7 Shows the quick fix that makes permutations of a method 30
3.8 The demo application for the (made up) electrical bill calculator running in the browser 31
3.9 The autocompletion GUI element. . . . 32
3.10 The variables listed by HDDebug. . . . 32

4.1 Four types of validation 36
4.2 The Chrome extension showing the value of the variables in the WHAP example. . . . 41

4.3 Chrome extension message passing. The gray node passes messages from the web page to the Chrome extension pop-up. Orange nodes are webpages that pass and receive messages to and from the message node. 42

5.1 The visualization the participants of the user study were given to build a constraint system in Task 7. 45

6.1 WebStorm suggestion for `className`. 53
6.2 WebStorm suggestion for a self-made component. 53
6.3 Snippets suggestions for making a component in WebStorm. 53
6.4 Inspection error showing a missing method, with a quick fix in WebStorm. 54
6.5 "LabVIEW State Machine example" by Smith Pohl, used under CC BY. 59

List of Tables

2.1 Variables in the electrical bill calculator. 8

5.1 Results from the user study. 49

C.1 All validation rules implemented in HDCode. 75

Listings

2.1 Temperature conversion between Celsius and Fahrenheit written in the HotDrink DSL. 11
2.2 Temperature conversion between celsius, fahrenheit and kelvin written in HotDrink DSL. 12
2.3 Temperature conversion between celsius and fahrenheit written in HotDrink API. 12
2.4 Example of binding together HTML input element to HotDrink variable. 14
2.5 Example of a custom React hook for binding together a React state with a HotDrink variable. 14
3.1 A sample of the HotDrink API code. 19
3.2 A sample of the HotDrink DSL code. 19
4.1 Code that implement scoping rules for allowing imported functions inside a method body. 34
4.2 Code for the implementation of validation rule for initializing all variables in a list of variable declarations to zero. 37
4.3 Code for the implementation of quick fix that initializing all variables in a list of variable declarations to zero. 37
4.4 Generating a Sprotty node for a constraint. 38
4.5 Code for the implementation of a code generator. 39
4.6 Adding the constraint system to the global `window` object. 41
4.7 The HotDrink DSL specification for the *Width-Height-Area-Perimiter* (WHAP) example. 41
5.1 JavaScript code given to participants in Task 3.1. 44
A.1 Langium grammar code for HotDrink DSL. 69
B.1 Binder functions for binding HTML elements and HotDrink variables, generated be the demo application feature. 72

Chapter 1

Introduction

Graphical user interfaces (GUIs) are an important part of everyday end-user interaction. GUIs of a wide range of applications often have one thing in common: they have dependencies between different elements of the GUI. These dependencies can be error-prone and complicated to manage, and every dependency that is added to the GUI increases the complexity of the GUI. It is difficult for a developer to keep track of all the dependencies in the GUI, and it is even more difficult to keep track of the many different ways the end user can interact with the GUI. This means that the GUI might easily end up in an invalid state, which can cause the GUI to crash or behave in an unexpected way. This is why constraint systems are important for GUI programming, they can be used to model GUI dependencies in a declarative way, and are less prone to error than traditional GUI programming. By using constraint systems, the developer can declare the dependencies as relations between the elements in the GUI, and the constraint system will ensure that the system keeps these relations satisfied at all times.

HotDrink [26, 28] is a JavaScript library for constructing multi-way dataflow constraint systems. It works by defining dependencies as constraints between elements on the GUI. HotDrink has been around for quite some time, but has not gotten much attention. We believe that this is because HotDrink has no dedicated tool support, and this is needed in modern software development. With tool support, we mean an Integrated development envirement (IDE), which is a textual editor that is aware of HotDrink and provides the developer with code completion, error highlighting, syntax highlighting, and other features that are common in modern IDEs. This is why we have created a IDE for HotDrink, for editing and debugging HotDrink based GUI specifications. We also include support for visual specifications. We implement these tools using the language

workbench Langium [8], visual framework Sprotty [15], and the extension Application Programming Interface (API) of Google Chrome [3].

To validate our implementation, we conducted a user study. The study adds confidence to our first hypothesis: an IDE could help a wider adoption of HotDrink. The results of the user study show that the IDE is useful, and that it is a good tool for developing HotDrink based GUI specifications. Even the participants with none to little experience with HotDrink were able to accomplish the tasks in the user study.

This thesis is organized as follows. In Chapter 2 we give a technical overview of constraint systems, the multi-way dataflow constraint system library HotDrink, and the architecture of Google Chrome extensions. The chapter also gives a brief description of the technologies used in the implementation. Chapter 3 gives an overview of the IDE—*HDCode*—and the Google Chrome extension—*HDDebug*—we have implemented in this thesis, and their individual features. Chapter 4 describes the technical implementation of the HDCode and HDDebug in depth. We introduce the user study in Chapter 5, and discuss the results from this study. In Chapter 6 we present related work, and finally in Chapter 7 we conclude the thesis and glance at possible future work to be done.

Chapter 2

Background

In this chapter, we present a technical definition of constraint systems. We then introduce *HotDrink*, a JavaScript library for defining and solving constraint systems. Finally, we explain how HotDrink Domain Specific Language (DSL) is transformed into HotDrink's JavaScript API code.

2.1 Constraint systems

We talk first about the *dataflow* of a program, which can be described as a graph of variables. The flow of data is expressed through directed edges between variables. If a variable x is involved in the computation of a variable y, then we say that the variable y is *dependent* on the variable x, and must therefore be recalculated every time variable x is changed.

A constraint system represents relations between one or more variables. We model a constraint system S as a tuple $S = \langle V, C \rangle$, where V is a set of variables, and C is a set of constraints, each of which is modeled as a tuple $\langle R, r, M \rangle$. A relation $R \subseteq V$ is the set of variables derived from the constraint, r is some n-ary relation among variables in R where $n = |R|$, and M is a set of constraint satisfaction methods [31].

$$S = \langle V, C \rangle, \quad C = \langle R, r, M \rangle, \quad R \subseteq V, \quad n = |R| \tag{2.1}$$

We consider in this thesis dataflow constraint systems, which are a kind of constraint systems. A dataflow constraint system allows for one or more constraint satisfaction *methods* to be associated with each constraint. These methods $m \in M$ in (2.1) describe how a constraint can be enforced. Each of these methods has a set of input and output variable(s), denoted by $\text{ins}(m)$ and $\text{outs}(m)$. We say that the constraint is satisfied, if the values of variables in R satisfy r. A constraint system can either be one-way or multi-way [42, 31]. This is determined by the dataflow between variables in the constraint.

A **one-way** dataflow constraint has exactly one method that can be used to satisfy it. In the equation, (2.2), below the computation of y depends on the product of values x_k, therefore, y is dependent on x_k. From this equation, we are only able to compute the value one way. There is no method to calculate a specific value of x_k, given the value of y. Hence, the constraint system is one-way.

$$y \leftarrow \prod_{k=1}^{n} x_k \tag{2.2}$$

On the other hand, a constraint can exhibit several different dataflows, in which case it is called a **multi-way** dataflow constraint. An example of this, is a computation of the variable x which is determined by the value of the variable y, while the computation of the variable y is determined by the value of the variable x. An example of this is shown in equations (2.3) and (2.4), therefore, this constraint system is multi-way.

$$y \leftarrow 2 \cdot x \tag{2.3}$$

$$x \leftarrow \frac{y}{2} \tag{2.4}$$

The following example, shows how a commonly known equation can be represented as a constraint. In equation (2.5), we have the relation between `celsius` and `fahrenheit`. A method that modifies the value of `celsius` whenever the value of `fahrenheit` is changed is generated by solving the equation in (2.5) for `celsius`, and can be seen in equation (2.6). Likewise, a method that modifies the value of `fahrenheit` whenever the value of `celsius` is changed is generated by solving the equation in (2.5) for `fahrenheit`, and can be seen in equation (2.7). We say that this constraint can be enforced in two ways, therefore it is a multi-way constraint.

$$\texttt{celsius} = \frac{\texttt{fahrenheit} - 32}{1.8} \tag{2.5}$$

$$\texttt{celsius} \leftarrow \frac{\texttt{fahrenheit} - 32}{1.8} \tag{2.6}$$

$$\texttt{fahrenheit} \leftarrow \texttt{celsius} \cdot 1.8 + 32 \tag{2.7}$$

We have now described an example of a simple constraint system, with only two variables—`celsius` and `fahrenheit`. Now we show where a constraint system helps in keeping relations between variables. In the example below, we introduce a (made up) electrical bill calculator. This system contains 15 variables with 7 constraints, which have 20 methods in total. In Table 2.1, we outline the variables in the system. All of these variables are dependent on each other; some are only dependent on one variable, while others are dependent on multiple variables, and some variables are dependent on each other. This is thus a multi-way constraint system. If we look at the equations bellow, we can see that the relations in this example, are more sophisticated than in the previous example. The values are more intertwined with each other, as showcased in Figure 2.1, and it is not straightforward to see what will happen if we change one of the variables. This is where a planning algorithm comes in play.

$$c_1 = \{x_{15} \leftarrow \frac{x_{11}}{x_{14}} \cdot 100, x_{11} \leftarrow x_{14} \cdot \frac{x_{15}}{100}, x_{14} \leftarrow \frac{x_{11}}{x_{15}} \cdot 100\}$$

$$c_2 = \{x_{11} \leftarrow x_9 \cdot x_{12} + x_{10} + x_4 + x_2, x_9 \leftarrow \frac{x_{11} - x_{10} - x_4 - x_2}{x_{12}},$$

$$x_4 \leftarrow x_{11} - x_2 - x_{10} - x_9 \cdot x_{12}, x_{12} \leftarrow \frac{x_{11} - x_{10} - x_4 - x_2}{x_9},$$

$$x_{10} \leftarrow x_{11} - x_2 - x_4 - x_9 \cdot x_{12}, x_2 \leftarrow x_{11} - x_4 - x_{10} - x_9 \cdot x_{12}\}$$

$$c_3 = \{x_4 \leftarrow 1000, x_5 \leftarrow x_4 \neq 0\}$$

$$c_4 = \{x_2 \leftarrow x_3 \ ?\ 0.5 : 0, x_3 \leftarrow x_2 \neq 0\}$$

$$c_5 = \{x_8 \leftarrow x_{11} - x_{11} \cdot \frac{x_6}{100}, x_{11} \leftarrow x_8 + x_8 \cdot \frac{x_6}{100}, x_6 \leftarrow \frac{x_{11} - x_8}{x_{11}} \cdot 100\}$$

$$c_6 = \{x_7 \leftarrow x_6 \neq 0, x_6 \leftarrow x_7 \ ?\ 1 : 0\}$$

$$c_7 = \{x_{12} \leftarrow f_1(x_1, x_{13}), x_{13} \leftarrow f_2(x_{12}, x_1)\}$$

A *planning algorithm* is used to compute the order in which the constraint methods are executed; this order is called a *plan*. This plan must ensure that once a variable has been read from, no other method can write to the variable; otherwise, constraints previously enforced could become invalid. We mention three planning algorithms, *DeltaBlue* [29], *SkyBlue* [36, 37], and *QuickPlan* [42], which all fall into the *incremental* category; they reuse previous plans to compute new plans.

The **DeltaBlue** algorithm is the baseline for the other two algorithms. The algorithm has two major limitations: there can be no cycles in either graph of constraints and variables, and each method can only have one output variable [29].

The **SkyBlue** algorithm was developed to overcome the limitations of the DeltaBlue algorithm—with support for cycles and multiple output variables [36, 37]. SkyBlue also

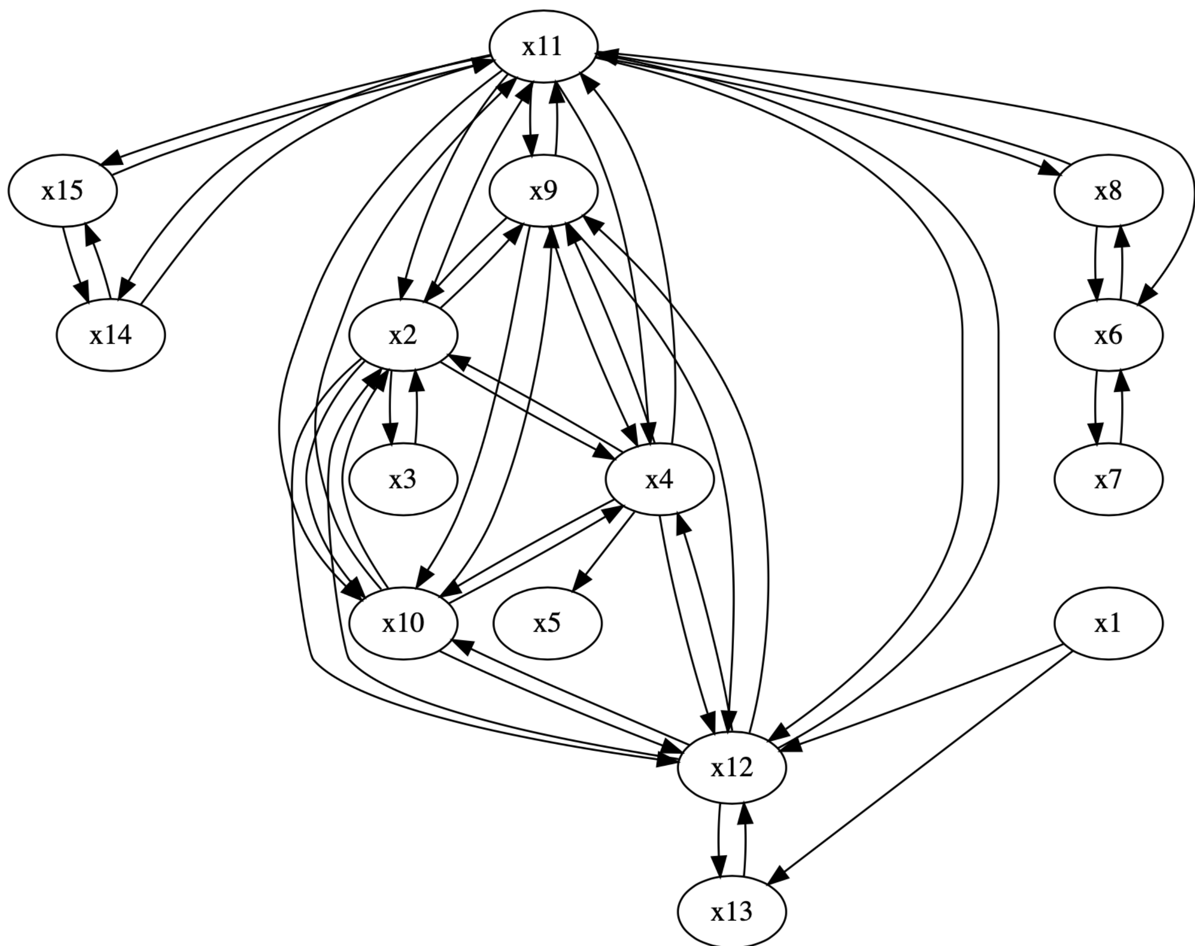


Figure 2.1: A graph showing the relations between the variables in the electrical bill example. Nodes represent variables and edges represent relations between variables. The Graphviz code for this graph, rendered by Graphviz, is generated by our tool based on the constraint system specification by our VSCode extension.

Variable name	Meaning	Type	Description	Relation
x_1	`place`	String	The power-region of Norway	c_7
x_2	`weekendFee`	Number	The extra cost of power in the weekends	c_1, c_4
x_3	`weekendFeeBool`	Boolean	If the bill has an extra cost when its a weekend, true if it is and false if it is not	c_4
x_4	`basePrice`	Number	A one time start price	c_2, c_3
x_5	`basePriceBool`	Boolean	If there is a start price, true if it is and false if it is not	c_3
x_6	`discountPercent`	Number	The percent of discount on the full price	c_5, c_6
x_7	`discountBool`	Boolean	If there is a discount, if true the discount is subtracted from the sum	c_6
x_8	`discountedPrice`	Number	The price after the discount is subtracted	c_5
x_9	`kWh`	Number	The number of kWh in the calculation	c_2
x_{10}	`serviceExtras`	Number	The cost of net service and rent of the power gird	c_2
x_{11}	`totalWithoutDiscount`	Number	The cost without subtracting the discount	c_1, c_2, c_5
x_{12}	`norwayBasePricekWh`	Number	The cost per kWh depending on where in the country the bill is calculated for	c_2, c_7
x_{13}	`pricekWh`	Number	The cost for one kWh	c_2, c_7
x_{14}	`salary`	Number	The user's salary	c_1
x_{15}	`percentSalary`	Number	The percentage of income the bill cost	c_1

Table 2.1: Variables in the electrical bill calculator.

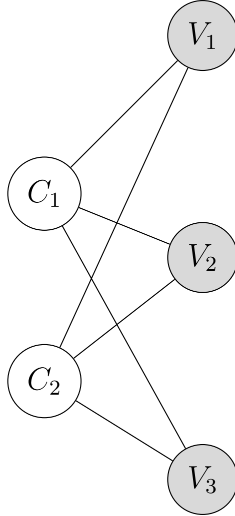


Figure 2.2: An example of an undirected bipartite graph.

has some limitations; it can not guarantee an acyclic solution, if a cyclic solution exists [42]. Then, in the worst-case it requires exponential time—$O(M^N)$ where M is the maximum number of methods per constraint, and N is the number of constraints [42].

Finally, the **QuickPlan** algorithm builds upon the SkyBlue algorithm. QuickPlan guarantees an acyclic solution—if there is one—and can satisfy the system in quadratic time $O(N^2)$. In practice it often allows for finding solutions in time less than $O(N)$[42].

As was earlier discussed and shown in Section 2.1, while a constraint system can be represented as a tuple, it can also be represented as an undirected bipartite graph [40]. In this graph, we denote each constraint and variable with vertices, and an edge between variable(s) and constraint(s), if the constraint is dependent on the variable. This is shown in the Figure 2.2; constraints are denoted as C_1 and C_2, while variables are denoted as V_1, V_2 and V_3. A representation often used by algorithms that operate on a constraint system, is called *constraint graph*. This is a more detailed version of the graph discussed above. An example of a constraint graph is shown in Figure 2.3, which is a visualization of the constraint system described in (2.8). In the constraint graph, instead of having constraint and variable vertices, we have constraint satisfaction method and variable vertices. In this graph, we have directed edges between the method inputs, ins(m), and the method, here, the flow of data goes from ins(m) to the method. We also have a directed edges between the method outputs, outs(m), and the method, accordingly, the flow of data goes from the method to outs(m).

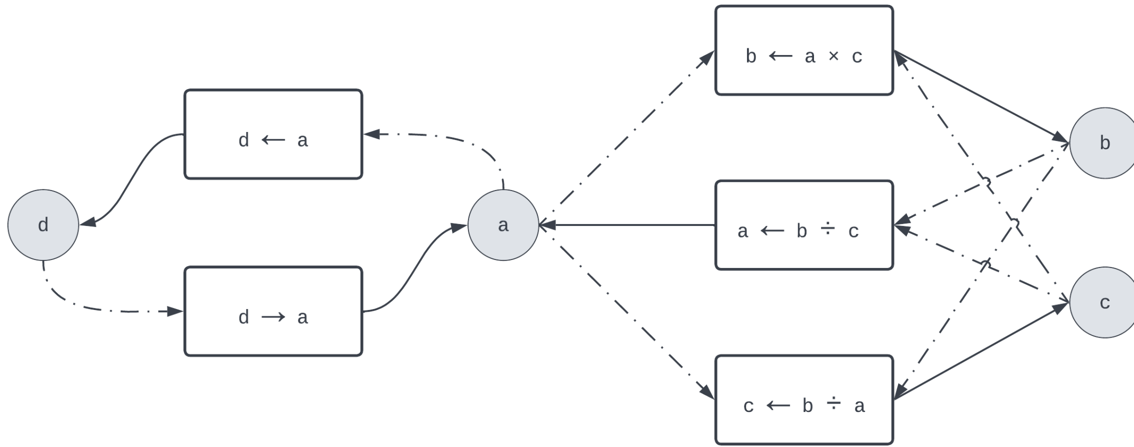


Figure 2.3: Example of a constraint graph.

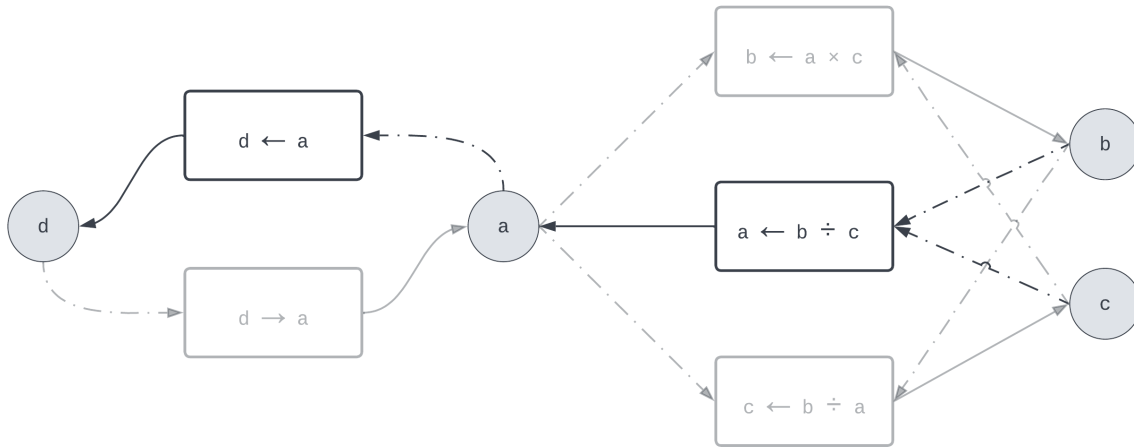


Figure 2.4: Example of a solution graph, non-greyed out element are chosen as the solution graph.

$$
\begin{aligned}
S = \langle V, C \rangle, \quad V = \langle a, b, c, d \rangle, \quad C = \langle c_1, c_2 \rangle, \\
c_1 = \langle \langle a, d \rangle, M_1 \rangle, \quad M_1 = \langle a = d, d = a \rangle, \\
c_2 = \langle \langle a, b, c \rangle, M_2 \rangle, \quad M_2 = \langle a = b/c, b = a \times c, c = b/a \rangle
\end{aligned}
\tag{2.8}
$$

To create a plan, we need to find a *solution graph*, this is a subgraph of the constraint graph. A solution graph must contain one method from each constraint, it has to be acyclic, and it can have at most one incoming edge to each variable. The solution graph is found by *planning* the constraint system. In Figure 2.4, we can se such a graph.

In some cases, we might have an overconstrained constraint system. This means that there are no valid solutions to the system, such as $\beta = \gamma$ and $\beta \neq \gamma$, hence, there is also no solution graph. On the other hand, we might have an underconstrained constraint system, which means that there are multiple valid solutions to the system. Therefore, we may have more then one solution graph [40].

2.1.1 HotDrink

In this thesis, we consider the multi-way dataflow constraint system library *HotDrink* [26]. HotDrink is a JavaScript library[1] written in FlowType—a typed variant of JavaScript, that compiles to JavaScript—that allows for creation of constraint systems. The fact that HotDrink is a JavaScript library makes it convenient for web development and the development of GUIs. Instead of writing explicit event handlers, the developer defines declarative data dependency specifications from which the library derives GUI behavior.

HotDrink allows one to specify *components*, *constraints*, *methods* and *variables*. Components are the top-most structure; they hold one or more variables and may or may not have constraint(s). A constraint within a component can only use the variables inside the same component. Methods work as earlier discussed in this section.

Listing 2.1: Temperature conversion between Celsius and Fahrenheit written in the HotDrink DSL.

```
1 let TemperatureConversion = component`
2  var celsius = 1337, fahrenheit;
3
4  constraint constr_degrees {
5    method_from_c_to_f(celsius -> fahrenheit) => celsius*9/5 + 32;
6    method_from_f_to_c(fahrenheit -> celsius) => (fahrenheit-32)*5/9;
7  }
8 `;
```

Constraint systems in HotDrink can either be specified using the HotDrink API or the HotDrink DSL. For example, from Section 2.1, we have the relation between `celsius` and `fahrenheit` that can be specified in these two ways, as shown in Listing 2.1 and 2.3. The first one (Listing 2.1) uses the HotDrink DSL, implemented using JavaScript tagged template literals[2]. The second one (Listing 2.3) uses the HotDrink API.

We can observe, in Listing 2.1, that the component, `TemperatureConversion`, has two variables, `celsius` and `fahrenheit`, with exactly one constraint, `constr_degrees`,

[1] https://git.app.uib.no/Jaakko.Jarvi/hd4

[2] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

that has two methods, `method_from_c_to_f` and `method_from_f_to_c`. As the naming suggests, the first method converts from celsius to fahrenheit, and the second method converts from fahrenheit to celsius.

Listing 2.2: Temperature conversion between celsius, fahrenheit and kelvin written in HotDrink DSL.

```
1  let TemperatureConversion = component`
2    var celsius = 1337, fahrenheit, kelvin;
3
4    constraint constr_c_f {
5     method_from_c_to_f(celsius->fahrenheit) => celsius*9/5+32;
6     method_from_f_to_c(fahrenheit->celsius) => (fahrenheit-32)*5/9;
7    }
8    constraint constr_k_f {
9     method_from_k_to_f(kelvin->fahrenheit) => kelvin*9/5-459.67;
10    method_from_f_to_k(fahrenheit->kelvin) => (fahrenheit+459.67)*5/9;
11   }
12 `;
```

In Listing 2.2 we expand the example from Listing 2.1. We add a new variable, `kelvin`, and with it we make a new constraint that contains two methods. The first method converts from kelvin to fahrenheit, and the second method converts from fahrenheit to kelvin. We see now that there are no methods that convert between `celsius` and `kelvin`, since the system will first calculate a value for fahrenheit, with the `method_from_c_to_f` from the `constr_c_f` constraint, and then use the method, `method_from_f_to_k` from the `constr_k_f` constraint to convert to `kelvin`. A visual representation of the constraint graph can can be seen in Figure 2.5, where the non-grayed out elements are the plan. The variables `celsius`, `kelvin` and `fahrenheit` are denoted by the letters `c`, `k` and `f`, respectively.

Listing 2.3: Temperature conversion between celsius and fahrenheit written in HotDrink API.

```
1  const system = defaultConstraintSystem;
2
3  const TemperatureConversion = new Component("TemperatureConversion");
4
5  system.addComponent(TemperatureConversion);
6
7  const celsius = TemperatureConversion.emplaceVariable("celsius", 1337);
8  const fahrenheit = TemperatureConversion.emplaceVariable("fahrenheit");
9
10 const method_from_c_to_f = new Method(2, [0], [1], [maskNone],
     ↪ (celsius) => (celsius * 9 / 5 + 32));
11 const method_from_f_to_c = new Method(2, [1], [0], [maskNone],
     ↪ (fahrenheit) => (fahrenheit - 32) * 5 / 9);
12
13 const constr_degreesSpec = new ConstraintSpec([method_from_c_to_f,
     ↪ method_from_f_to_c]);
14
15 const constr_degrees =
     ↪ TemperatureConversion.emplaceConstraint("constr_degrees",
     ↪ constr_degreesSpec, [celsius, fahrenheit], false);
```

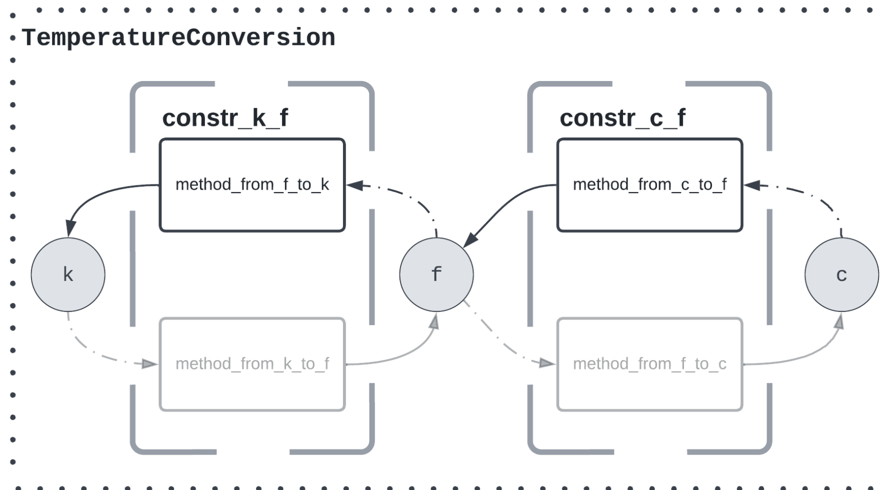


Figure 2.5: Plan for the constraint system in Listing 2.2. The non-grayed out vertices and edges are the solution graph.

In Listing 2.3, we define an empty constraint system on line 1, create a component on line 3, and add the component to the constraint system, on line 5. On line 7, we declare the variable `celsius` and give it the initial value of `1337`. Next, on line 8, we declare the variable `fahrenheit` without any initial value. Further down, on line 10 and 11, we declare the methods `method_from_c_to_f` and `method_from_f_to_c`, respectively. The first method, `method_from_c_to_f`, converts `celsius` to `fahrenheit`, and the second method, `method_from_f_to_c`, converts `fahrenheit` to `celsius`. The `Method` constructor takes five arguments: the number of variables the method interacts with, an array of indexes to the input variable(s), an array of indexes to the output variable(s), an array of `MaskType`[3], and a function that the method should execute. We now create, on line 13, the specification of the constraint, where we pass an array that contains the two methods from line 10 and 11 as an argument. Finally, on line 15, we define the constraint `constr_degrees`. The `emplaceConstraint` method takes four arguments: the name of the constraint, the specification of the constraint, from line 13, an array containing the variables references, from lines 7 and 8, and at last a boolean value that indicates if the constraint is optional. The array of variables references, given as the third argument, contains the variables that the constraint interacts with. The index of the variable in this array is the index of the variable for the second and third argument in the method

[3]MaskType is a number that specifies the type of the corresponding input variable. The MaskType `maskNone` (`0`) describes a basic datatype, `maskPromise` (`1`) describes that the input variable is a promise and the method has to wait for that to resolve before the method can be executed, `maskUpdate` (`2`) describes that the input variable is a reference to a object, `maskPromiseUpdate` (`3`) is a combination of `maskPromise` and `maskUpdate`.

constructor seen on line 10 and 11.

The Model-View-ViewModel (MVVM) [30] design pattern separates the development of the GUI—the *view*—the development of the business logic—the *model*—and the *view-model*. The view-model can be described to hold a state of the data in the view. A *binder* serves as a communication channel between the view and the view model. HotDrink fulfills the role of the view-model in MVVM. The view and the model themselves are outside the scope of HotDrink [28]. Thus, whenever a variable value is updated in the view, the view notifies the binder, and the binder updates the corresponding variable in HotDrink with that value. HotDrink, on the other hand, will send an event to the binder when a variable in the constraint system is updated. Then the binder updates the corresponding variable in the view with that value.

In Listings 2.4 and 2.5, we show an example of a binder for HTML and React [10], respectively.

Listing 2.4: Example of binding together HTML input element to HotDrink variable.

```
1  function binder(element, value, type) {
2      value.value.subscribe({
3          next: val => {
4              if (val.hasOwnProperty('value')) {
5                  element[type] = val.value;
6              }
7          }
8      });
9      element.addEventListener('input', () => {
10         value.value.set(element[type]);
11     });
12 }
13
14 export function valueBinder(element, value) {
15     binder(element, value, "value");
16 }
```

Listing 2.5: Example of a custom React hook for binding together a React state with a HotDrink variable.

```
1  export function useHDBinding<T>(hdValue: HDValue<T>): [T, (newValue:
       ↪ T) => void] {
2      const [value, setValue] = useState<T>(hdValue.value);
3
4      useEffect(() => {
5          hdValue.subscribe({
6              next: (val: any) => {
7                  if (val.hasOwnProperty('value')) {
8                      setValue(val.value)
9                  }
10             }
11         });
12     }, [hdValue]);
13
14     return [value, (newValue: T) => hdValue.set(newValue)];
15 }
```

2.2 Architecture of Google Chrome extensions

Google Chrome [3], one of the most popular web browsers, features a framework for developing extensions. An extension is an application that may alter the browser's behavior. It is usually constructed by one or more HTML and JavaScript files. The extension runs in its own environment, separated from the main browser thread, and thus cannot access elements from the webpage. Chrome relies on *content scripts* to interact with the webpage.

A content script is a JavaScript file that is injected into the webpage, so that it runs in the context of the webpage. The script is running in the same render process as the webpage itself, and can access the resources of the webpage, like the Document Object Model (DOM) and global variables. The content script and the extension can communicate with each other by sending messages, using the chrome API. Google Chrome works by allowing one browser kernel process to operate in privileged mode; this process has access to platform and system resources, therefore, render processes request resources from the kernel process [34]. Each rendering process may be linked to a web page, and these processes run in separate environments.

2.3 Technology stack

Our goal is to enable developers to effectively write and debug HotDrink code. To achieve this goal, an IDE tailored specifically for HotDrink is needed. Such an IDE should provide a wide range of code services, such as syntax-aware code editing and syntax highlighting, code completion, navigation, formatting, and folding. The IDE should also perform static code analysis to a certain extent, and report validation errors, provide "quick fixes" (which are proposals to solve an issue in the code), as well as provide rich code visualization facilities, and debugging functionality. It should be possible to transpile HotDrink DSL specifications into target code from within an IDE and create application templates.

Implementing an IDE from scratch is time-consuming and error-prone. We use a *language workbench*, which is a tool that allows for designing and implementing new software languages and IDEs for them. We give an overview of language workbench functionality in Section 6.4.

In this thesis, we use the language workbench Langium [8] to implement the IDE for HotDrink. For that, we specify the grammar for the language, the validation rules and quickfixes for the language, visualization, and code generation. Langium builds a parser based on a grammar. The parser converts the input string into a representation of a semantic model. The visualization is done by using the Sprotty framework [15]—a web-based diagramming framework by Eclipse Foundation. IDEs output by Langium are integrated with Language Server Protocol (LSP) [35], which is a standardised protocol, developed by Microsoft[4], for communication between a text editor or an IDE and a language server. The LSP provides a wide range of language support features, such as code completion, syntax highlighting, go to definition, and type-checking Therefore, we obtain all these features if the IDE is built to support the LSP.

The use of Langium allows us to use any IDE that supports LSP. We chose to use Visual Studio Code (VSCode) as our IDE because it is a popular IDE and it is free and open source. In addition to this, Langium also has direct integration with the VSCode extension API[5]. Langium outputs an extension for VSCode which supports the HotDrink DSL, and is available for download on the VSCode marketplace. An extension is an added feature for the editor; in this case the HotDrink extension adds HotDrink DSL support to VSCode, with the features mentioned earlier. This makes it easy for HotDrink developers to use HotDrink in the well-established VSCode editor.

Apart from the textual specification of HotDrink DSL, we also devised a visual representation of the constraint system implemented with HotDrink. To implement it, we used the *Sprotty framework* [15], which is a web-based diagramming framework that provides integration with the LSP; hence, it can be implemented together with Langium. A constraint system specification is rendered as a graph; this choice of representation is motivated by the fact that this data structure is well-known among a wide audience of developers.

To simplify the graph representation, we decided to merge constraint and component into one node type. This node then has edges to the constraint satisfaction method(s). Methods and variables are nodes with directional edges. The input-variable(s) of a method are connected to the method with an edge, where the flow is from the variable to the method. Thus, the output-variable(s) of the method are connected with an edge where the flow is from the method to the variable.

[4]https://microsoft.github.io/language-server-protocol/
[5]https://code.visualstudio.com/api/language-extensions/overview

Developers also need to be able to debug HotDrink code. The first step to achieving this is to be able to see the current value of a HotDrink variable. To do this, we implemented a Google Chrome extension that is aware of HotDrink variables and can display their current values. The extension works as an observer that monitors the value of HotDrink variables and displays them in a popup window. This is especially useful in HotDrink specifications where some of the values are not displayed as their own GUI element, such as an indexing of an autocompletion menu selection. In such an example, it can be hard to see what is causing a problem right away, thus, an extension showing the value of all HotDrink variables can be very useful.

Chapter 3

HDCode and HDDebug

Programming GUIs is a significant part of all development effort. Programming GUIs is also a relatively complex task. Input fields of different types, such as text boxes, checkboxes, drop-down lists, and so on, frequently appear in GUIs. All of these various types of input fields tend to result in intricate relationships between the input fields. As discussed in Section 2.1.1, the HotDrink library can be used to model these relationships in a less intricate way. This thesis work builds tooling for implementing such models effectively. In this chapter we describe our toolset—HDCode and HDDebug—which are extensions to VSCode and Google Chrome, respectively.

The VSCode extension provides the following functionality:

- Support for the HotDrink DSL.
- Generating HotDrink API JavaScript code from the HotDrink DSL specification.
- Generating a demo application from the HotDrink DSL specification, which contains an HTML page and JavaScript code that uses the HotDrink API, and binds the view and view-model [30].
- Visualization of the constraint system specified by the HotDrink DSL.

The Google Chrome extension enables showing the current value of all variables in the HotDrink specification.

These tools are meant for developers to use the HotDrink library in a productive way. These tools help developers adopt HotDrink into their projects, and also makes HotDrink more attractive to them.

3.1 HDCode

Below we showcase two different ways of writing a HotDrink DSL specification. In Listing 3.1 we see a HotDrink specification written in plain JavaScript, using the HotDrink API directly. Programming such specifications is a very error-prone process, and finding and fixing code errors a significantly difficult process. In larger systems, it is easy to get lost in all the constraints, not knowing which variables are affected by other constraints. In Listing 3.2 we see a HotDrink specification written in the HotDrink DSL, using JavaScript tagged template literals. This way of writing HotDrink is the recommended way[1], since the code is more compact and easier to read, but it is still an error-prone and difficult process to find and fix code errors. For developers to be able to write HotDrink specifications in a more productive and less error-prone way, we have developed HDCode. As mentioned earlier in this chapter, HDCode is an extension to VSCode that provides the following features: support for the HotDrink DSL, generation of JavaScript code for the HotDrink API, generation of a demo application, and visualization of the constraint system.

Listing 3.1: A sample of the HotDrink API code.

```
1  const system = defaultConstraintSystem;
2  const myComponent = new Component("MyComponent");
3
4  system.addComponent(myComponent);
5
6  const x = myComponent.emplaceVariable("x");
7  const y = myComponent.emplaceVariable("y", 1);
8
9  const method1 = new Method(2, [0], [1], (x) => x + 1);
10 const method2 = new Method(2, [1], [0], (y) => y - 1);
11
12 const calculateSpec = new ConstraintSpec([method1, method2]);
13
14 const calculate = myComponent.emplaceConstraint ("calculate",
       ↪ calculateSpec, [x, y], false);
```

Listing 3.2: A sample of the HotDrink DSL code.

```
1  let MyComponent = component`
2      var x, y = 1;
3
4      constraint calculate {
5          method1(x -> y) => x + 1;
6          method2(y -> x) => y - 1;
7      }
8  `;
```

[1]It is worth noting that the HotDrink DSL is transpiled, in the background, to HotDrink API code.

Figure 3.1: Auto-completion of a component in HDCode.

3.1.1 Support for the HotDrink DSL

The first feature of HDCode is support for the HotDrink DSL notation. In VSCode the developer can make a file with the file extension `.hd`. In this file, the developer can write HotDrink DSL: *components* with *variables* and *constraints*, and *methods* inside the constraints, like we describe in Section 2.1.1. Variables support the data types `string`, `number` and `boolean`. The developer can also optionally import functions, to be used in the methods, from other JavaScript files, as described in Section 4. When keywords, such as `import`, `component`, `var`, `constraint` and `method`, are typed, they are highlighted. In Figure 3.1, we show the syntax highlighting of the keywords `component` and `constraint` of the HotDrink DSL using the HDCode extension.

Figure 3.1 shows the auto-completion that is aware of the HotDrink DSL syntax. Inside a `.hd` file, a developer can start typing `c` in an empty file and the auto-completion will suggest `component` in a drop down menu, below the typed spot. This works for all the keywords, and for variable suggestion inside a `method`. The auto-completion is aware of the semantic model, and will suggest the correct keywords, according to the context, i.e, if the developer is typing inside a `component`, the auto-completion will suggest `var` and `constraint`, but not `method` or `component`. Alternatively, the developer my press `Ctrl + space`, to get the auto-completion menu at any time. These features helps to make it more intuitive for the developer to write HotDrink using the DSL, and easier to write semantically and syntactically correct code.

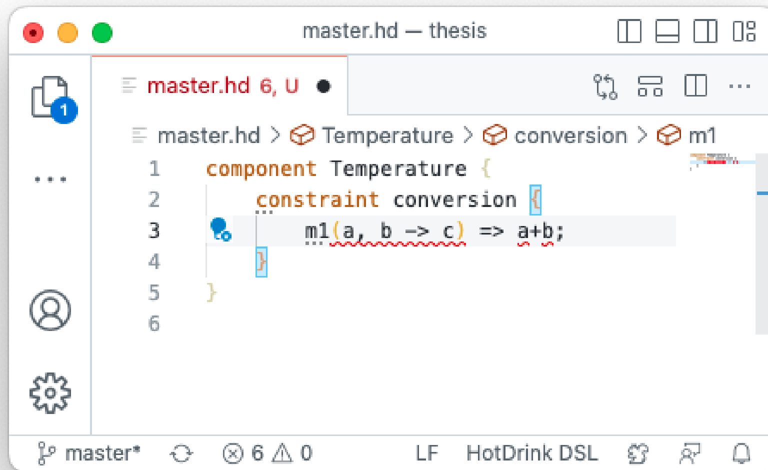


Figure 3.2: Validation of a HotDrink DSL specification in HDCode. The developer is using variables that are not defined, which is an error, hence, the underlined syntax is in red.

While writing a HotDrink DSL specification, the extension will provide the developer with validation of the syntax, and checking of some semantic errors. Showcased in Figure 3.2. The extension will underline the syntax in red if there is an *error*, in yellow if there is a *warning*, and in blue if there is a *info*. The developer can hover over the underlined syntax to get a description of what is the reason for the underlining.

The extension also has a few code actions, called *quick fixes*, that can be applied to the underlined syntax with a hover button at the start of a line; this is called a *hint*. The developer can hover over the underlined syntax, or press `cmd + .`, to get a list of the available quick fix(es) for the current hint. When a quick fix is executed, it alters the content of the file, in a way that the hint is no longer present.

The developer can navigate to the definition of a *variable* from everywhere in the file using `cmd + click`. Variables can also be renamed, and the renaming will be reflected in all the places, where the variable is used.

3.1.2 JavaScript code for the HotDrink API

As the second feature, the developer can generate JavaScript code using the HotDrink API from the HotDrink DSL specification. To do this, the developer can press `cmd`

+ `shift + p` to open the command palette. This is an interactive menu in VSCode that allows the developer to search for commands and execute them. The developer can then type `HotDrink: Generate JavaScript code`, and press `enter` to execute the command. The extension will then generate a JavaScript file, with the same name as the `.hd` file, in a folder called `generated`, in the same directory as the `root` folder. The Figures in 3.3 shows this process. Figure 3.3a shows the folder structure before the command was executed, and Figure 3.3b shows the folder structure after the command was executed.

In this JavaScript file, shown in Figure 3.3c, only the component(s) are exported, hence the developer can import the component(s), in other JavaScript files. Thus, the developer can not access the variables, or the methods, or the constraints, from the JavaScript file. This is done to prevent the developer from ever directly interacting with the HotDrink API, hence it is less prone to errors. The extension uses the validation features to let the developer know if there are any errors in the HotDrink DSL specification. If the developer tries to execute the function from the command palette, VSCode will also let the developer know, through an error message window in the bottom right corner[2], and the function will not be executed and the file will nor be made. For this file to work in an application, the npm-package of `hotdrink`[3] needs to be installed.

3.1.3 Demo application

To showcase new HotDrink developers the capabilities of HotDrink, we implemented a command to make a *demo application*. The demo application is a basic web application, which contains one HTML file and three JavaScript files.

The generation of the demo application is done in the same way as the generation of the HotDrink API code, described in Section 3.1.2, except that the developer types `HotDrink: Generate demo application` in the command palette. The command will generate the four files previously mentioned. The content of these files depends on the HotDrink DSL specification. The HTML file is a basic HTML file, without any styling (Cascading Style Sheets (CSS), Syntactically Awesome Style Sheets (Sass), Leaner Style Sheets (Less) etc.). An input field will be created for each variable in the specification[4]: boolean variables create an input element with type checkbox, number variables create an input element

[2]This is the standard way of showing error messages inside VSCode.

[3]`https://www.npmjs.com/package/hotdrink`

[4]A prerequisite for making a demo application is that all variables are typed, with types described in Section 3.1.1

(a) Before generating the JavaScript code.

(b) After generating the JavaScript code.

(c) HotDrink API code generated from the HotDrink DSL specification.

Figure 3.3: Generation of HotDrink API code from a HotDrink DSL specification.

with type number, and string variables create an standard input element. In total, three JavaScript files are generated; the HotDrink API code (modified to work in an demo application), the *binders* for binding the view—HTML—to the view-model [30]—HotDrink API—as mentioned in Section 2.1.1, and finally the usage of these binders to bind the view variable(s) to the corresponding view-model variable(s).

To run the application, the developer need to start an HTTP server; one alternative of doing this is to use a Python HTTP server, by navigating to the root folder, and executing the command `python3 -m http.server <port>`. An other alternative is to use the VSCode extension `Live Server`[5], which is an extension that starts a HTTP server, and updates the browser when the content of the files change. When the server is running, we can interact with the the HotDrink specification through the HTML file. Values will change according to the constraints specified in the specification.

For HotDrink to work in the application, a compiled version of the HotDrink library needs to be present in the application. The path of the HotDrink file needs to correlate with the script source tag in the HTML file, and with the JavaScript file where HotDrink is imported.

3.1.4 Visualizing HotDrink specifications

To make it easier for the developer to understand HotDrink DSL specifications, and follow the dataflow of the constraint system, our tool supports visualization features. We decided to show the HotDrink DSL specification as a graph. This graph view can be seen in Figure 3.4, beside the corresponding HotDrink DSL specification. In this graph green nodes represents a constraint, the node is labeled with the name of the component and constraint, like `<component>.<constraint>`. From a component/constraint node grayed out undirected edges with a red dot in both ends connect it to the methods. Methods are blue nodes, labeled with the name of the method. These nodes have directed edges to the input variables and output variables, respectively. The direction of these edges goes from the input variables to the method node, and from the method node to the output variables. The variables we denote as yellow nodes, labeled with the name of the variable.

The graph opens in a separate webview in VSCode. When the webview is open, the developer can interact with the graph, by clicking and moving on the nodes and edges. The view is also dynamic, thus changes in the HotDrink DSL specification will be reflected

[5]`https://marketplace.visualstudio.com/items?itemName=ritwickdey.LiveServer`

Figure 3.4: Graph view of a HotDrink DSL specification that converts between the temperatures in degrees Celsius and Fahrenheit.

in the graph view. The developer can also zoom in and out, by scrolling with the mouse wheel, and move the graph, by clicking and dragging the mouse. This is especially useful when the graph is large and the developer wants to see the whole graph or just a part of the graph.

We also added a feature to let the developer see the relations between the variables, as a graph. Below, in Figure 3.5b, we see an example of this graph. To generate this graph, one can press `cmd + shift + p`, to open the command palette. Then navigate to the `HotDrink DSL: Generate sourcefile for web graph` command. When this command is executed, a new file is generated in the `/generated` folder, with the name `<component-name>Digraph.txt` (Figure 3.5a). The content of this file is a DSL for a web graph generator, called Graphviz Online[6]. The Graphviz Online tool can generate a graph, based on the DSL we generated.

3.1.5 HDCode in action

We now demonstrate the features of the extension with the example from Table 2.1, the made up electrical bill calculator. In a VSCode IDE with the extension installed, we open a new folder, in it we make a new file called `electrical_bill.hd`, and start writing the HotDrink DSL specification.

[6] `https://dreampuf.github.io/GraphvizOnline/`

whapDigraph.txt — thesis

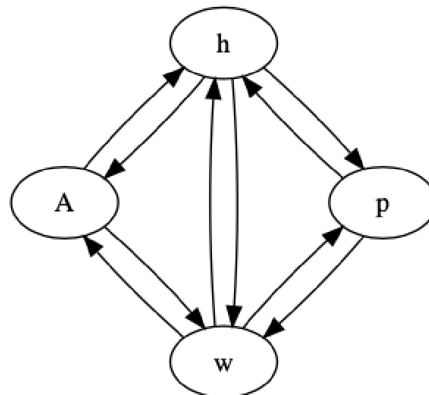
EXPLORER
OPEN EDITORS
master.hd U
whapDigr... U
THESIS
generated
whapDigraph.... U
master.hd U
OUTLINE

whapDigraph.txt U
generated > whapDigraph.txt

```
1  digraph whap {
2      w -> A;
3      h -> A;
4      p -> w;
5      h -> w;
6      A -> w;
7      p -> h;
8      w -> h;
9      A -> h;
10     w -> p;
11     h -> p;
12 }
```

master* ⊗ 0 ⚠ 0 ✓ Spell Prettier

(a) The code for the Graphviz Online DSL generated by the extension.



(b) The rendered graph.

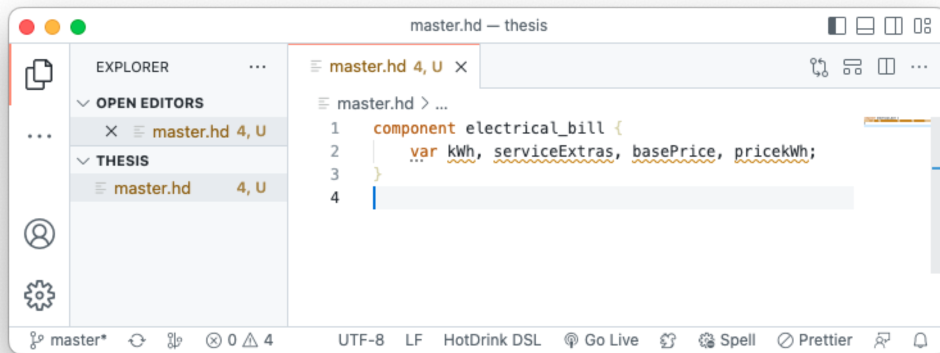
Figure 3.5: Variable relations visualized as a graph.

In an empty file we are only allowed to start typing one of two keywords—`import` or `component`—to see the whole grammar see Appendix A. We start typing `component`, right away the IDE gives us a suggestion to complete the word for us, we press `enter` to accept the suggestion. Now we need to give the component a unique identifier, in this case `electrical_bill`. We press `Ctrl + Space`, to see the two types of allowed keywords inside a component—`var` (variables) and `constraint`—we want to add the variables we are going to use in the component first. All the lists of variable declaration start with the keyword `var`, followed by a unique identifier, and an optional colon and the type of the variable. First we define the list of variables that should be given initial value, as shown in Figure 3.6a. Then we look at the hint, and see that we are able to use a quick fix: the quick fix sets the type for all variables in the list of variable declarations to `number`, and initializes all the variables in the list with the value `0`, shown in Figure 3.6b, and the result of the quick fix can be seen in Figure 3.6c. From this we can change the value `0` to the intended value. Then we declare the rest of the variables with there respective types.

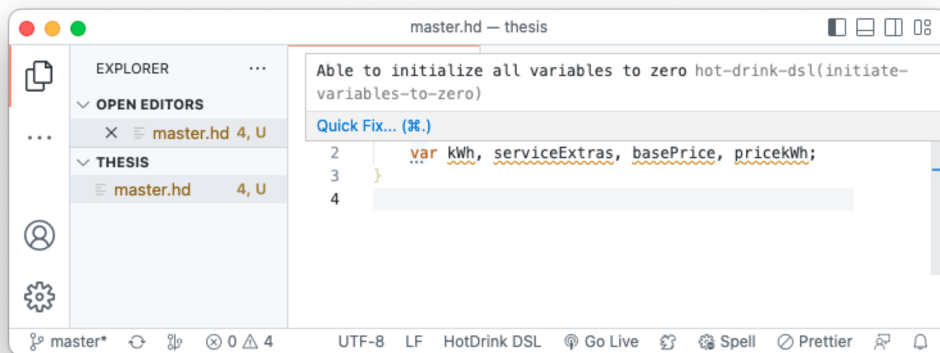
At this point, we declared made the component and variables, and we can start defining the constraints with their respective methods. We start with the constraint, `constr_total`. This is the biggest constraint with six methods and six variables in total. The methods in the `constr_total` constraint uses simple math for the calculation of their output variables, and therefore we write the calculation of these methods directly in the HotDrink DSL specification. All methods in a constraint need to use the same set of variables, the extension will give a error over the hole constraint, if one of the methods in the constraint uses a variable that is not in all other methods, or if one of the methods are missing one or more variable(s) that the other methods are using. This way the extension can make sure that the constraint is valid, and that the developer is not missing any variables, or using variables that are not in the constraint's scope.

The next step is to write the constraint `constr_salary`, we write the first method, `(x, y -> z) => y / x * 100`, but we give the method the same name as the first method in the constraint `constr_total`, this is allowed, but we get a warning that the method names need to be unique for the demo generation to work, hence we change the name of the method to a unique one.

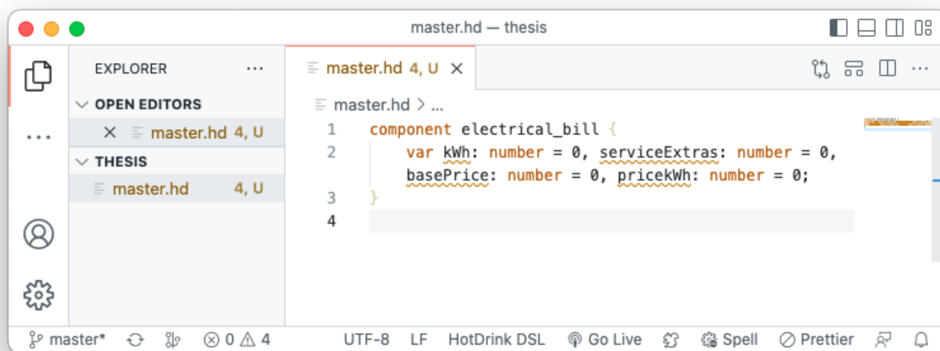
Then we hover over a *hover button*, indicated by three dots (...) at the start of the line (an example of these three dots can be seen in Figure 3.6a, on line 2 under the `var` keyword) to see the *hint* and, and press `Ctrl + Space`, to use the corresponding quick fix *"Make permutations"*. This will manipulate the text in the document and add five new methods with temporary string values as the body. The developer has to remove the



(a) List of variable declarations inside the `electrical_bill` component.



(b) Showing the quick fix for initializing all variables in a list of variable declaration to zero.



(c) Showing the manipulation done by the quick fix.

Figure 3.6: Initiation of variables using a quick fix

once that are not needed and write the correct body for the methods that are needed. This can be seen in Figures 3.7.

At this point we need to make a constraint that have methods that uses more than simple math. We make a different JavaScript file for the functions for the method bodies. Then inside the HotDrink DSL specification, we press `Ctrl + Space` over the component, and select the import statement. Inside a set of curly brackets we write the name of the functions we want to import, followed by the keyword `from`, and the path to the file where methods are located. Now we are able to use this function inside the body of the methods, in our HotDrink specification file.

We now make the rest of the constraint to make the made up electrical bill calculator work, and we are done with the HotDrink DSL specification. To see the whole specification as a graph we can either use the command pallette, or click on the graph icon in the status bar, both will open the graph view. Nodes and edges can be moved around and zoomed in and out to make the graph easier to read and understand.

If we want to see the relation between the variables in the constraint system just made, we can do this by using the command pallette, and selecting the command `HotDrink DSL: Generate sourcefile for web graph`. This will generate a file in the `/generated` folder, with the name `<component-name>Digraph.txt`. In this file we are provided with the DSL for a web graph generator, called Graphviz Online[7]. We get the graph earlier shown in Figure 2.1.

To see the constraint system in action, we can use one of the commands that comes with the extension to generate a demo application. We open the command pallette once more, and type `HotDrink DSL: Generate demo application`, and press `enter`. This process is described in Section 3.1.3. When the process is done, we can see and manipulate the generated demo application, to see the constraint system in action. Figure 3.8 shows the demo application running in the browser.

[7]`https://dreampuf.github.io/GraphvizOnline/`

(a) Showing the constraint before the quick fix.

(b) Showing the manipulation done by the quick fix.

(c) The result after manually removing unnecessary methods, made by the quick fix, and the method body written for the methods left.

Figure 3.7: Shows the quick fix that makes permutations of a method

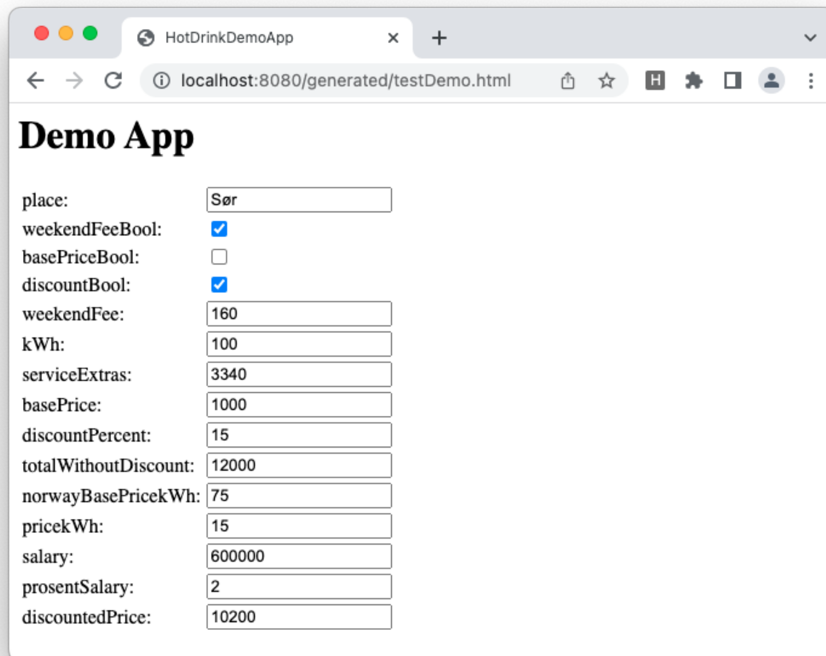


Figure 3.8: The demo application for the (made up) electrical bill calculator running in the browser

After testing out the system in the demo application we can use the last command—`HotDrink DSL: Generate JavaScript from the current HotDrink DSL file`—to generate the HotDrink API code, as described in Section 3.1.2 ready to be used in a production JavaScript application, with the npm package `hotdrink`[8].

3.2 HDDebug

Often in GUI applications, the user can interact with input-fields of various types, and the GUI will perform operations on the input, and display the result in the GUI. Sometimes the GUI have variables that do not have theire own input-field, we will call these *hidden variables*, but are used in the calculation of elements in the GUI. These variables are hard to debug, and often a source of bugs in an application. Hence, we have developed a tool that can be used to debug these hidden variables (together with visible variables), and make it easier to find bugs in the application.

[8]`https://www.npmjs.com/package/hotdrink`

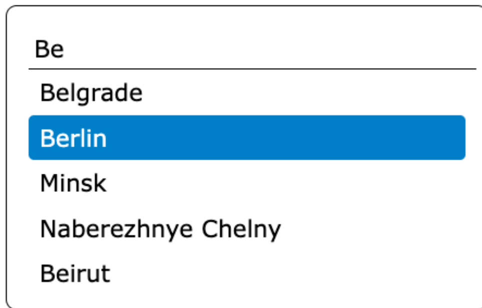


Figure 3.9: The autocompletion GUI element.

Figure 3.10: The variables listed by HDDebug.

We will now describe a simple example where we have a GUI element with a input-field When the user types in the field, a dropdown menu shows up with autocompletion suggestions. In this GUI element, we have four variables: `query`, `index`, `menu`, and `value`. Of these, `query`, `menu`, and `value` are shown in the GUI, thus `index` is a hidden variable. The `query` variable is the value of the input field, the `index` variable is the index of the selected item in the dropdown menu, the `menu` variable is the list of items in the dropdown menu, and the `value` variable is the value of the selected item in the dropdown menu. We have no way of knowing the value of `index`. Hence, we use HDDebug to list out the constraint system variables, and their values. By doing this we are able to see if the anomaly is caused by the hidden variable. Figure 3.9 shows the GUI element, and Figure 3.10 shows the variables listed in HDDebug.

Chapter 4

Implementation

We have implemented IDE support for the multi-way dataflow constraint system HotDrink, and a Google Chrome extension for showing the values of the variables of the constraint system. In this chapter we describe our implementation[1].

4.1 Integrated Development Environment

Our implementation uses the language workbench Langium [8], which provides an IDE that supports the development of HotDrink. We decided to use Langium as our language workbench as it is an actively developed language workbench, and outputs an extension for VSCode—a popular IDE for JavaScript development. As a language workbench, Langium provides us with tools to define our language, scope our language, validate the language, manipulate the file the language is defined in, visualize the language, and generate code for General Purpose Languages (GPL). Langium also allows for testing the defined language, the validation, the manipulation, and generated code. This all starts with defining the grammar of our DSL in the Langium grammar language, which uses an extended Backus-Naur form (EBNF) like syntax, similarly to Xtext [25] grammars. From this grammar, Langium provides us with a parser, and a semantic model that Langium uses to generate the Abstract Syntax Tree (AST). This semantic model is written in TypeScript[2], hence the AST has type safety. This helped in the development process, as we could make changes to the grammar, and the type safety would follow through the generated parser and semantic model. A possible disadvantage of using Langium is that it is not as mature as other language workbenches.

[1]The implementation can be found at `https://github.com/MathiasSJacobsen/HotDrink-DSL` and `https://github.com/MathiasSJacobsen/chrome-hotdrink-extension`.

[2]`https://www.typescriptlang.org/`

4.1.1 Grammar

We decided to replicate the grammar of the HotDrink DSL as closely as possible. The grammar can be seen in Appendix A. The grammar allows for making all the key features of HotDrink, such as the use of components, variables, constraints and methods. Where the implemented grammar differs from the HotDrink grammar is inside a method body: we only allow for simple math and boolean expressions, such as addition, subtraction, multiplication, division and logical operators. In the HotDrink grammar, the method body can contain JavaScript in function bodies. To compensate for this, we added the ability to import functions to our grammar and use the imported functions in the method body. We decided to do this, as we wanted to keep the grammar as simple as possible, hence we did not want to implement the grammar of JavaScript in our grammar.

4.1.2 Scoping

A method are only able to read from the input variable(s), thus we had to implement a scoping rule that only shows input variable(s) in the autocompletion menu inside a method body. This was done by overwriting the default scoping rule of Langium and adding a new scope for the method body. The same had to be done for the imported methods to be used in the method body. The scoping rule can be seen in Listing 4.1.

Listing 4.1: Code that implement scoping rules for allowing imported functions inside a method body.

```
1  export class HotDrinkDslScopeProvider extends DefaultScopeProvider {
2    descriptionProvider: AstNodeDescriptionProvider;
3
4    getScope(node: AstNode, referenceId: string): Scope {
5      ...
6      if (referenceId === "FunctionCall:funcRef") {
7        const modelNode = getContainerOfType(node, isModel);
8        const descriptions = modelNode!.imports
9          .flatMap((importStm) => importStm.imports)
10         .map((_import) => {
11           if(_import.altName) {
12             return this.descriptionProvider.createDescription(_import,
       ↪ _import.altName.name, getDocument(_import))
13           }
14           return this.descriptionProvider.createDescription(_import,
       ↪ _import.function.name, getDocument(_import))
15         });
16       return new StreamScope(stream(descriptions));
17     }
18     ...
19 }
```

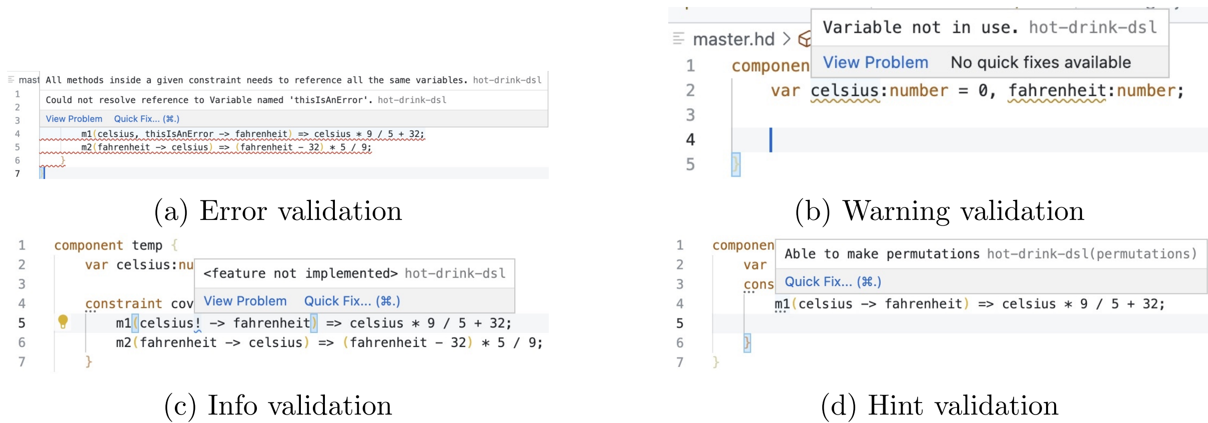
4.1.3 Validation and manipulation

To showcase the capabilities of Langium, we implemented the use of validations in the IDE, in this Section we will mention some of these validation rules. To see all validation rules and a more detailed description of these rules, see Appendix C.

Validation rules come in four *severity levels*:

- **Error** — The validation rule underlines the error in red (see Figure 4.1a). This validation has to be fixed for the code to be in a valid state. In our case, errors are often used to uphold the HotDrink preconditions, such as the input variables of a method are the only variables that can read from in the method body, all methods in the constraint must use the same set of variables, and so on.
- **Warning** — The validation rule underlines the warning in yellow (see Figure 4.1b). Warnings are optional to fix, the code can be in a valid state with or without fixing the warning. In our implementation, warnings are used to notify the developer that some functionality of the extension might not work as intended in the current state, and remind that elements of code should be unique. For example, methods need to have unique names. If they do not, the code generation will not work. Warnings that a variable is not in use are supported too.
- **Info** — The validation rule underlines the info in blue. Inforules are for showing more information to the developer; the code is valid. We use info validation when the programmer tries to use functionality that is not jet implemented, see Figure 4.1c.
- **Hint** — The validation rule differs from the other three types. Instead of underlining words, the start of a line comes with a button that when hovered over shows the validation rule. This can be seen in Figure 4.1d. The main purpose of this rule is to hook it up with a *quick fix*. We implemented hints, mainly to showcase the abilities of the extension, with the ability to make permutations from one method in a constraint, the ability to initiate all variables in a list of variable declarations to zero, and so on.

Quick fixes manipulate the content of file, and can be linked to a validation rule. In our implementation, as mentioned, we use them together with the *hint* validation rule. These are powerful tools, as they can access the AST of the file and use it to manipulate the content of the file. To do this, we specify a range in the file we want to manipulate, and the content we want to replace the range with.



(a) Error validation

(b) Warning validation

(c) Info validation

(d) Hint validation

Figure 4.1: Four types of validation

In the Listings 4.2 and 4.3, we show how the hint to initialize all values in a list of variable declaration is implemented, and how the quick fix to that validation is implemented, respectively. The validation rule is implemented in the `HotDrinkDslValidator` class. Every validation takes two arguments: the AST node to validate and a `ValidationAcceptor` function. The `ValidationAcceptor` takes three arguments: the severity of the validation (error, warning, info, or hint), the text shown in the dialog box when the validation is triggered, and a `DiagnosticInfo` object where the information about the rule is kept. This object has one obligatory field, and eight optional fields (displayed in italic):

- `node` — The AST node that triggered the validation.
- *`property`* — A property of the AST node that triggered the validation. If given the validation will be restricted to this property.
- *`index`* — If the property is a array, the index can specify which element in the array the validation should be restricted to.
- *`range`* — If the validation should be restricted to a range in the file, this field can be used to specify the range, instead of a property.
- *`code`* — A code that can be used to identify the validation, also shown in the User Interface (UI).
- *`codeDescription`* — A description of the error.
- *`tags`* — Metadata about the diagnostic.
- *`relatedInformation`* — Related information about the diagnostic.
- *`data`* — A data entry field. This field is preserved between a `textDocument/publishDiagnostics` notification and `textDocument/codeAction` request.

The quick fix is implemented in the `HotDrinkDslActionProvider` class. The methods for

each quick fix take two arguments: a diagnostics object that represents diagnostics found in the compiler, such as errors or warnings, and the Langium document that holds the AST of the file and any state that is derived from the AST. A quick fix is intertwined with a validation rule by the `code` property in the `DiagnosticInfo` object, seen on line 11 in Listing 4.2. Then we use the `data` property to pass information needed for the quick fix, such as the indices of the component and variable declaration, as seen i Figure 4.2. The range property on line 16 is used to specify the range in the file that should be replaced, and this range is derived from the `range` property in the `DiagnosticInfo` object, and the `newText` property. On line 17 is the content that should replace the range.

Listing 4.2: Code for the implementation of validation rule for initializing all variables in a list of variable declarations to zero.

```
1  export class HotDrinkDslValidator {
2    ...
3    hintToInitializeVariablesToZero(
4      vars: Vars,
5      accept: ValidationAcceptor
6    ): void {
7      if (vars.vars.every(varRef => varRef.initValue === undefined)) {
8        accept("hint", "Able to initialize all variables to zero", {
9          node: vars,
10         property: "vars",
11         code: IssueCodes.InitiateVariablesToZero,
12         data: vars.$container.$containerIndex?.toString() + "." +
               ↪ vars.$containerIndex?.toString()
13       })
14     }
15   }
16   ...
17 }
```

Listing 4.3: Code for the implementation of quick fix that initializing all variables in a list of variable declarations to zero.

```
1  export class HotDrinkDslActionProvider implements CodeActionProvider {
2    ...
3    private initiateVariablesToZero(
4      diagnostic: Diagnostic,
5      document: LangiumDocument
6      ): CodeAction {
7      const range = diagnostic.range;
8      const model = document.parseResult.value as Model;
9      const indexes = (diagnostic.data as string).split(".").map(v =>
         ↪ parseInt(v));
10     const names =
         ↪ model.components[indexes[0]].variables[indexes[1]].vars.map(v
         ↪ => v.name);
11     return {
12       title: 'Initiate variables to zero',
13       kind: CodeActionKind.QuickFix,
14       diagnostics: [diagnostic],
15       isPreferred: true,
16       edit: {
17         changes: {
18           [document.textDocument.uri]: [{
19             range,
20             newText: `var ${names.join(": number = 0, ")}: number = 0;`
21           }]
```

```
22            }
23        }
24      };
25    }
26    ...
27  }
```

4.1.4 Visualization

In Section 2.1 we describes ways of visualizing constraint systems as graphs. Now we will describe how we implemented another graph visualization of a constraint system. As mentioned in Section 2.3, we decided to use the *Sprotty framework* [15] to implement the graph visualization. The framework opens a new tab element in the IDE, and the graph is rendered inside the tab as a webview.

Each node is composed by the same elements: list of children, type, id and layout options. In the list of children we add a label, this label is attached to a node as a text to be displayed on the node. The type of the node describes what the node is, such as a variable, a method, a constraint, etc. The color of the node is also determined by the type. The id is a unique identifier that is used to identify the node in the graph. The layout options are used to specify styling and position for the node in when it is rendered in the webview. In Listing 4.4 we see how we generate a node for the constraint.

Listing 4.4: Generating a Sprotty node for a constraint.

```
1  protected generateConstraintNode(
2    constraint: Constraint,
3    { idCache }: GeneratorContext<Model>
4    ): SNode {
5    const name = constraint.name ? constraint.$container.name + '.' +
       ↪ constraint.name : constraint.$container.name + '.' + NO_NAME;
6    const nodeId = idCache.uniqueId(name, constraint);
7
8    return {
9        type: 'node:constraint',
10       id: nodeId,
11       children: [
12           <SLabel>{
13               type: 'label',
14               id: idCache.uniqueId(nodeId + '.label'),
15               text: name
16           }
17       ],
18       layout: 'stack',
19       layoutOptions: {
20           paddingTop: 10.0,
21           paddingBottom: 10.0,
22           paddingLeft: 10.0,
23           paddingRight: 10.0
24       }
25   };
26 }
```

We decided to make every component its own graph. Therefore, if there are more than one component in the constraint system, there will be more than one separate graph in the webview. We chose not to provide components with their own nodes; instead we labeled the constraint nodes with the name of the component, followed by a dot and then the name of the constraint. This makes it easy to see what nodes are connected to what component, in the case of more than one component, and also helps making the graph short and concise. From these components/constraint nodes, undirected edges are drawn to the method nodes, to show what methods are inside the constraint. The method nodes are then connected to the variable nodes, with directed edges. A input variable has a directed edge pointing from the variable node to the method node, while a output variable has a directed edge pointing from the method node to the variable node.

We decided to make the three different types of nodes (components/constraints, methods and variables) to have different colors. The first—component/constraint nodes—is colored green, the second—method nodes—is colored blue, and the third—variable nodes—is colored yellow.

4.1.5 Code generation

We wanted developers to be able to test the constraint system they are building, thus developers have to be able to import the constraint system into their own code base, and to be able to visualize variable relations. Therefore, as discussed in Section 3.1, our implementation features several code generators. These work by accessing the DSL model, and then extracting information needed for the code generation, such as the name of the components, the names of the methods, the bodies of the methods, etc. In Listing 4.5, we show how we generate the code for the web graph. We first extract the information needed, then we compose the content of the file, at last we write it to the file.

Listing 4.5: Code for the implementation of a code generator.

```
1  export function generateWGraph(
2    model: Model,
3    filePath: string,
4    destination: string | undefined
5  ) {
6    const data = extractDestinationAndName(filePath, destination);
7    const generatedFilePath = `${path.join(
8      data.destination,
9      model.components[0].name + "Digraph.txt"
10   )}`;
11   const fileNode = new CompositeGeneratorNode();
12
13   const graph = makeGraph(model);
14   const entries = Object.entries(graph);
15
```

```
16    fileNode.append(`digraph ${model.components[0].name} {`, NL);
17
18    entries.forEach(([key, value]) => {
19      if (value.size !== 0) {
20        value.forEach((v) => {
21          fileNode.append(`${v} -> ${key};`, NL);
22        });
23      }
24    });
25
26    fileNode.append("}", NL);
27
28    if (!fs.existsSync(data.destination)) {
29      fs.mkdirSync(data.destination, { recursive: true });
30    }
31    fs.writeFileSync(generatedFilePath, processGeneratorNode(fileNode));
32  }
```

Our implementation incudes code generation for the following:

- **Generate JavaScript from the current HotDrink DSL file** — Generates a JavaScript file that contains the HotDrink API code, that reflects the HotDrink specification in the HotDrink file. The file is meant to be used with the `HotDrink npm-package`.
- **Generate Demo** — Generates a full demo that follows the MVVM [30] design pattern, described in Section 2.1.1. For the view-model we generate a JavaScript file with the HotDrink API (this file uses a compiled version of HotDrink library). To represent the view, a HTML file is generated, containing a input-field for each variable in the view-model. For the view and the view-model to communicate, we generate two JavaScript files—the first contains the binder functions (see Appendix B to see content of this file), and the second uses the functions from the first file to bind the view and model.
- **Generate sourcefile for Graphviz** — Generates a file containing the relation between the variables with the DSL for the Graphviz Online. This tool will generate a visualization of these relations as a graph.

4.2 Chrome extension

In a web application, one GUI element can contain many different variables, not all of these have there own input-/show-field. Therefore, we implemented a Chrome extension that injects a script onto a web page—that uses HotDrink—that shows the value of every variable in the constraint system. The extension can be used with the *demo application*

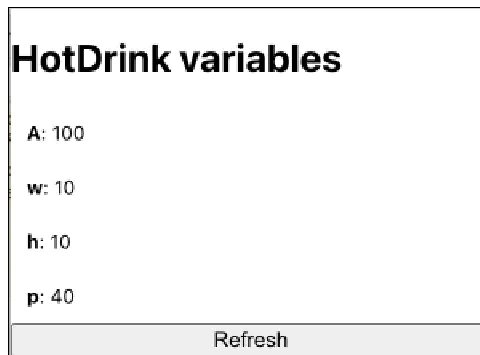


Figure 4.2: The Chrome extension showing the value of the variables in the WHAP example.

from the VSCode extension, without any configurations, other than installing the extension. For the extension to work on a web page that uses HotDrink, the web page must add the constraint system to the global window object. How we do this in the VSCode extension can be seen in Listing 4.6. The extension can be seen in Figure 4.2, running alongside a webpage containing a HotDrink constraint system, this system can be seen in Listing 4.7.

Listing 4.6: Adding the constraint system to the global `window` object.

```
1 const system = new hd.ConstraintSystem();
2 window.constraintSystem = system;
```

Listing 4.7: The HotDrink DSL specification for the *Width-Height-Area-Perimiter* (WHAP) example.

```
1  import { mySqrt } from './myMath.js'
2
3  component whap {
4      var A:number=100, w:number, h:number, p:number;
5
6      constraint Pwh {
7          m1(w, h -> p) => 2 * (w + h);
8          m2(p, w -> h) => p / 2 - w;
9          m3(p, h -> w) => p / 2 - h;
10     }
11
12     constraint Awh{
13         n1(w, h -> A) => w * h;
14         n2(A -> w, h) => [mySqrt(A), mySqrt(A)] ;
15     }
16 }
```

Our Chrome extension is separated into three parts: content script, background service, and a React app. *The content script* runs in the context of the current web page in the browser, and injects a script onto this page. This script extracts the constraint system from the `window` object, and posts a message—containing the extracted values—on the render thread, therefore, the background service can listen for messages sent from the

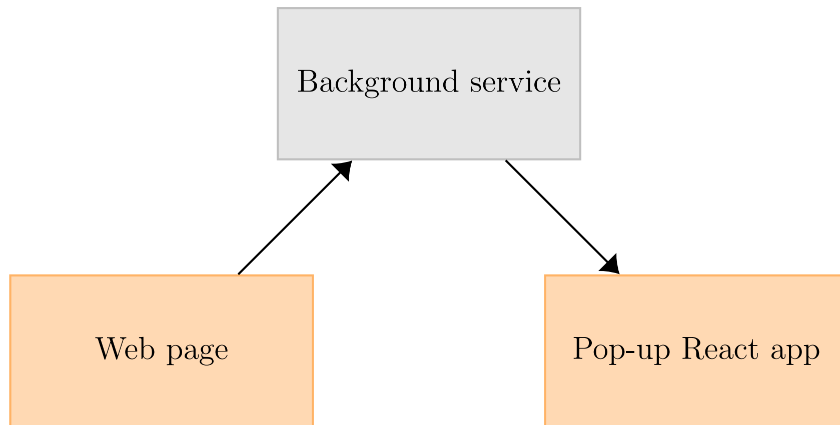


Figure 4.3: Chrome extension message passing. The gray node passes messages from the web page to the Chrome extension pop-up. Orange nodes are webpages that pass and receive messages to and from the message node.

web page. *The background service* is a JavaScript file that listens for the messages sent from the content script, whenever a messages arrives, the background service attaches the updated values to its own `window` object. From here *the React app* can extract these values, through the Chrome extension API [3]. The React app is a pop-up window that is shown when the extension icon is clicked, hence the application is its own webpage inside the browser.

Chapter 5

User study

We preformed a user study where we wanted to study the usability of the VSCode extension, how developers use various features in the extension, and how the extension helped them in the development process. We asked ten developers to participate in the study and provided them with a brief introduction to what HotDrink is, as well as on how to write HotDrink DSL code. This together with the VSCode extension main page on the VSCode marketplace[1] were the only information they had during the study. We also provided them with all the mathematical formulas needed for the tasks.

The participants where asked to complete six tasks that were designed to study all aspects of the extension. Each task was aimed at studying a specific feature, and the participants were asked to complete the tasks in the order they were presented. The participants was encouraged to speak their thoughts out loud and describe their thought process. In addition to the questions listed below, we also asked the participants to rate their own programming experience, on a scale from 1 to 5, where 1 is intro level programming course (at a university level or similarly) and 5 is professional developer. We also asked whether they had any prior knowledge of HotDrink. The tasks we wanted the participants to complete were:

1. Create a file with the HotDrink extension, and implement a temperature conversion between Celsius and Fahrenheit.

    1.1. Create a component.

[1] `https://marketplace.visualstudio.com/items?itemName=MathiasSkallerudJacobsen.HotDrink-DSL`

   1.2. Create the variables.
   1.3. Create a constraint.
   1.4. Implement the methods.

2. Generate a demo.

   2.1. Start a http-server.
   2.2. Calculate three different temperatures.

3. Add Kelvin to the temperature conversion, use the functionality of the extension to do this.

   3.1. Use the given JavaScript function (Listing 5.1) to convert from Kelvin to Celsius.
   3.2. Remake the demo and calculate three different temperatures.

4. Rename the variables using rename refactoring.

5. Generate JavaScript code from the HotDrink specification.

   5.1. Describe what in the JavaScript code represent a constraint.
   5.2. Describe what in the JavaScript code represent a variable.
   5.3. Describe what in the JavaScript code represent a method.
   5.4. Describe what in the JavaScript code represent a component.
   5.5. Describe where the function body that calculates the temperature from Celsius to Fahrenheit is.

6. Use the visualization tool to visualize the constraint system.

7. Based on a given visual specification of a constraint system (as shown in Figure 5.1) write a corresponding HotDrink DSL specification.

   7.1. Create a component.
   7.2. Create the variables.
   7.3. Create a constraint.
   7.4. Implement the methods.
   7.5. Import JavaScript methods.

Listing 5.1: JavaScript code given to participants in Task 3.1.

```
1 export function kelvinToCelsius(kelvin) {
2     return kelvin - 273.15;
3 }
```

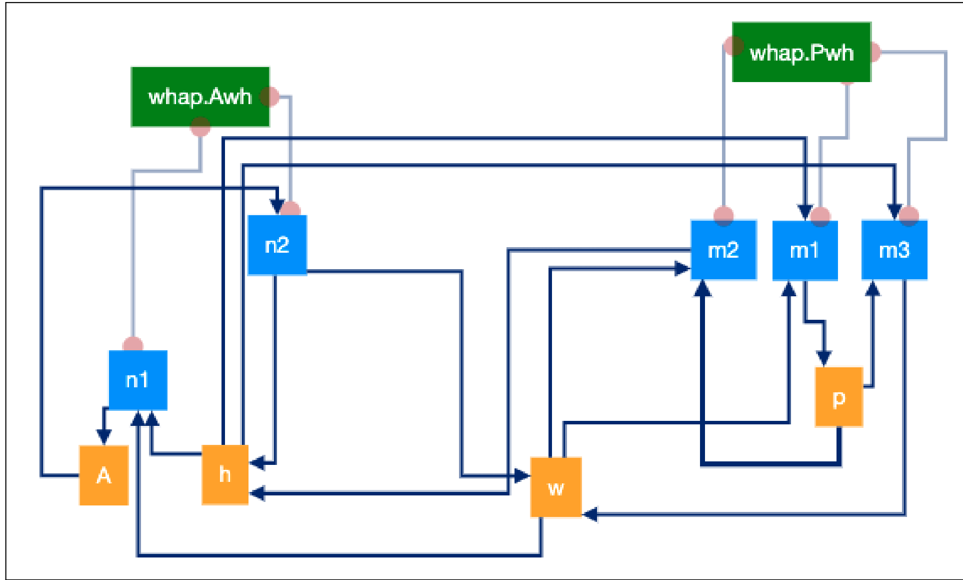


Figure 5.1: The visualization the participants of the user study were given to build a constraint system in Task 7.

5.1 Results

The results from all ten participants can be seen in Table 5.1. The two first columns (gray) are information about the participants; the first column is on a scale from 1 to 5, previously described, and the second column is either yes (☑) or no (☐). The rest of the table is how the participant preformed on each task. Green (●) means that the participant completed the task, yellow (◐) means that the participant completed the task with some guidance, and red (○) means that the participant did not complete the task correctly.

5.1.1 Task 1

This task aimed at studying how the syntax highlighting and keywords completion helped the developers. All participants completed the task, the common mistake was to forget to name the methods, which is not required in HotDrink, but is required in the VSCode extension for the generation to be used. We also noticed that the participants that had prior knowledge of HotDrink had a tendency to use the HotDrink syntax, instead of the VSCode extension syntax; they did not typeset the variables and they named the constraints and methods. They expressed that they knew that this naming is not required in HotDrink, so they wanted to write the code as they would write it in HotDrink.

Participants often used autocompletion for keywords when they where unsure what to write, and they also used syntax highlighting to see if they had written the code correctly. Some participants were confused by the similarly of a JavaScript lambda function and the method syntax, but with the help of some validation errors they figured it out.

5.1.2 Task 2

In this task we wanted to study how the participants used the demo generation, and their thoughts about the demo. All participants completed the task but one had some trouble with some JavaScript configuration that was not related to the extension.

5.1.3 Tasks 3 and 4

As discussed in Section 4.1.1, we did not want to reinvent the JavaScript grammar, hence Task 3 is aimed at how the participants used our implementation of using imported statements, instead of writing JavaScript in the body of the methods. Task 4 was aimed at studying how the participants used the rename refactoring.

On Task 3 for the participants to add Kelvin to the temperature conversion, they had to make a new constraint that handles the conversion between Kelvin and Celsius, each constraint updates one variable at a time which executing a plan. Instead of making a new constraint, most of the participants made a new method inside the previously made constraint. When doing this, all participants got a validation error (all validation errors are listed in Table C.1). From this error one of two things happened; either the participant figured out that a new constraint had to be made, or the participants added the new variable to the old methods. No participants had any problems with using the imported statement.

After four participants had completed the user study, we figured out that for the refactoring tool used by VSCode to work, the caret cannot be at the end of an identifier. We thus had to ask the participants to have the caret at the start of the identifier before trying to refactor.

5.1.4 Task 5

In this task we wanted to study if the participants could figure out what the generated JavaScript code correlated to in their HotDrink specification. On this task all participants did as expected, most of the participants took their time and read the hole file and noticed the different part of the file, while others rushed through the task and made some mistakes. On Task 5.5, where the participants had to describe where the function body that calculates the temperature from Celsius to Fahrenheit is, many of the participants at first point at the method line, but with some help, they realized that the function body was one of the arguments to the method constructor.

5.1.5 Tasks 6 and 7

At this point in the user study, we wanted to study how the participants used the visualization tool from a specification they had made, but also when they where given a graph to make a specification from. In Task 6, all participants could see what the different nodes where and how a method interacted with the different variables. While in Task 7 we gave them a graph, which can be seen in Figure 5.1, to make a specification from. In general, all the participants could make a specification from the graph, but some had some trouble, that perhaps can be blamed on programming experience.

5.2 Summary

All participants, both skilled and unexperienced with HotDrink, were able to completed the study. When first utilizing the extension, inexperienced HotDrink developers made fewer mistakes than experienced HotDrink developers. The strictness of the extension, as opposed to the less strict HotDrink DSL that they were using, might be the explanation.

Developers with little experience will likely still have a hard time using HotDrink due to the complicated nature of the library. We believe that developers that know more about the fundamentals behind programming, can adapt easier to new syntax and aspects of programming. Most of the participants struggled with Task 3; this is because they did not understand the fundamental concept of a constraint, and that (in this temperature conversion example) it can only update one variable at a time. One can see that some experienced HotDrink developers also made mistakes on Task 3.

All in all, our study shows that the extension is useful for developers that either know or want to get to know HotDrink. It also shows that the experience level of the developer plays a somewhat significant role, but that the extension is still useful for both experienced and inexperienced developers.

Table 5.1: Results from the user study.

	Participant 1	Participant 2	Participant 3	Participant 4	Participant 5	Participant 6	Participant 7	Participant 8	Participant 9	Participant 10
Programming experience	★★☆☆☆	★★★☆☆	★★★☆☆	★★★★☆	★★★★☆	★★★★☆	★★★★☆	★★★★☆	★★★★★	★★★★★
Knowledge of HotDrink	☐	☐	☐	☐	☐	☐	☑	☑	☑	☑
1	●	●	●	●	●	●	●	●	●	●
1.1	●	●	●	●	●	●	●	●	●	●
1.2	●	●	●	●	●	●	◐	●	●	◐
1.3	●	●	●	●	●	●	◐	●	●	●
1.4	●	◐	◐	●	●	◐	◐	◐	◐	●
2	●	●	●	●	●	●	●	●	●	●
2.1	●	●	●	●	●	●	●	●	●	●
2.2	●	●	●	●	●	○	●	●	●	●
3	◐	◐	●	◐	◐	◐	●	◐	●	◐
3.1	●	●	●	●	●	●	●	◐	●	●
3.2	○	◐	◐	●	◐	◐	●	◐	●	◐
4	●	○	●	◐	●	●	●	●	●	●
5	●	●	●	●	●	●	●	●	●	●
5.1	○	●	●	●	●	○	●	●	●	●
5.2	●	●	●	●	●	●	●	●	●	●
5.3	●	●	●	●	●	●	●	●	●	●
5.4	●	●	●	●	●	●	●	●	●	●
5.5	◐	○	◐	●	●	◐	●	●	●	●
6	●	●	●	●	●	●	●	●	●	●
7	●	●	●	●	●	●	●	●	●	●
7.1	◐	●	○	●	●	●	●	●	●	●
7.2	○	●	●	●	●	●	●	●	●	●
7.3	●	●	●	●	●	●	●	●	●	●
7.4	○	●	●	●	●	●	●	●	●	●
7.5	●	●	●	◐	●	●	●	●	●	●

Chapter 6

Related work

6.1 Integrated development environments

IDE is a software suite that combines the fundamental tools needed to create and test software. Such programs often have development-related features such as text editors, compilers, test platforms, etc. These features are packed into one single stand-alone GUI. This makes it so the developer does not have to use different UIs for text editors, compilers, test platforms, etc. individually. The toolkit of an IDE is designed to simplify the development-process for the developer.

In the same way that there are different types of developers (frontend, backend, testers, full stack), there are different IDEs for different development environments. One can categorize IDEs in to four different types: *non-language-specific*, *language-specific*, *web-based*, and *cloud-based*. None-language-specific IDEs are those that are not bound to one single (or a small set) programming language(s). These IDEs are usually used for general purpose development, and often come with a limited set of features—syntax highlighting, autocomplete, etc.—and then it is for the developer to customize the IDE to their needs, with *plugins/extensions* made by the community. VSCode [12] and Atom [4] are examples of non-language-specific IDEs. Language-specific IDEs are those that are bound for a single programming language or a small set of common languages. These IDEs are usually used for language specific development, and come with all the tools needed for the developer to create, compile and test their code. Examples of language-specific IDEs are PyCharm [9] and Eclipse[1]. Web-based IDEs are used for development in HTML and

[1] `https://www.eclipse.org/ide/`

JavaScript. Web-based IDEs often come with a toolkit specialized for web development. VSCode [12] and Eclipse Orion[2] are examples of web-based IDEs. Cloud-based IDEs are offered in a Platform as a Service (PaaS) model. This makes it so that development tools are available wherever the developer wants to develop. This also removes compatibility issues, since the code is managed in the cloud. An example of cloud-based IDEs is AWS Cloud9 [5].

6.1.1 Visual Studio Code

Visual Studio Code [12] is a non-language-specific IDE developed by Microsoft, and is currently the most popular IDE according to a recent study[3]. The IDE is open source, web-based, and highly customizable through a large pool of community made extensions. These extensions make use of the API provided by Visual Studio Code, and allows extensions to interact with Visual Studio Code itself. These extensions can add features such as syntax highlighting, commands, themes, etc. It is based on the Electron framework [6].

6.1.2 Atom

Atom is a cross platform IDE developed by Github [4]. One of the key features of this IDE is the facilitated interaction with Github, and the git UI package it comes with. In Atom, plugins/extensions are called *packages*. These add features and functionality to the IDE. Atom is made with HTML, CSS, and Node.js, and it runs on Electron [6]. This makes it easy to alter the look and feel of the IDE with CSS, and add new features to the IDE with HTML and JavaScript.

6.1.3 PyCharm

PyCharm is an IDE developed by JetBrains [9]. PyCharm is made for writing Python code, but is also supports CoffeeScript, TypeScript, Cython and SQL, among others. For web-developing PyCharm offers framework-specific support for frameworks such as Django, Flask, Google App Engine, Pyramid, and web2py. PyCharm has tool support

[2] `https://projects.eclipse.org/projects/ecd.orion`

[3] `https://survey.stackoverflow.co/2022/#section-most-popular-technologies-integrated-development-environment`.

such as navigation compatibility, code completion, code analysis, quick fixes, and syntax highlighting. The IDE also comes with integrated tools such as a debugger, test runner, database tools and Version Control System (VCS) support. PyCharm also integrates with scientific tools such as IPython Notebook, matplotlib and NumPy, to mention a few.

6.1.4 Eclipse IDE

Eclipse IDE is developed by the Eclipse Foundation [2]. The IDE supports Java development, but can be configured with plug-ins to support languages such as C/C++, Python, Perl, etc. Even though Eclipse is somewhat specific to Java, this can make Eclipse a non-language-specific IDE. Eclipse IDE is open source, and it is widely used in the educational sector.

6.1.5 AWS Cloud9

AWS Cloud9 is a cloud-based IDE developed by Amazon [5]. It operates as a PaaS, that offers an IDE with a full set of tools needed for development, such as a code editor, debugger, code completion, linting, etc. The IDE integrates with other AWS through a built in terminal, which makes it easy to interact with the AWS ecosystem. Cloud9 is, as mentioned in Section 6.1, a cloud-based IDE. Hence a developer can ac¡cess the IDE from anywhere with a web browser. It comes pre-packed with tools for programming languages JavaScript, C++, PHP, Go, among others. The cloud-based IDE also lets the developers run the code in the cloud. Another major feature of Cloud9 is the ability to seamlessly code together with other developers in real time.

6.2 Support for web framework in integrated development environments

6.2.1 React in WebStorm

React [10] is a JavaScript library for building UIs. React works by rendering *components* to the DOM. When a state in that component changes, React re-renders only that component, instead of the entire DOM. React uses its own syntax extension of JavaScript

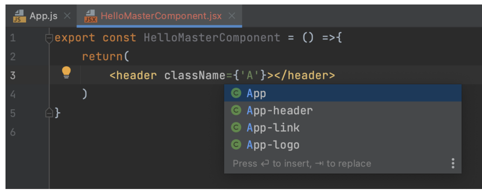


Figure 6.1: WebStorm suggestion for `className`.

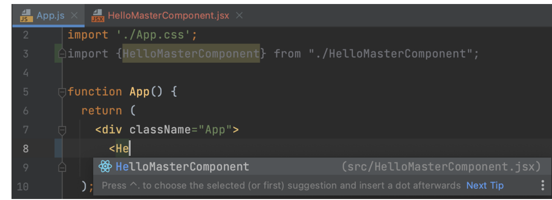


Figure 6.2: WebStorm suggestion for a self-made component.

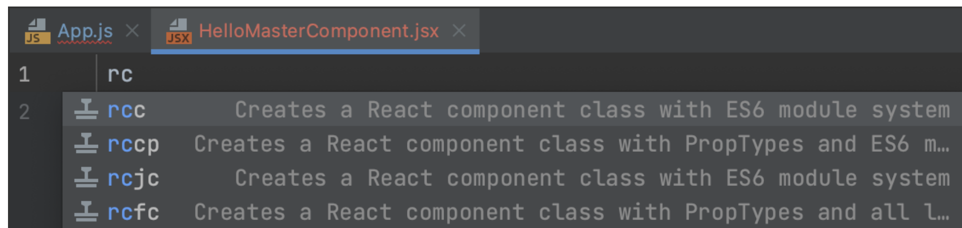


Figure 6.3: Snippets suggestions for making a component in WebStorm.

called *JSX*, which is a mix between HTML and JavaScript (optionally TSX can be used to get TypeScript support).

WebStorm provides the developer with code completion suggestions throughout the coding process. While the developer types in WebStorm, the code completion panel will popup with suggestions for the developer to choose from. It will suggest React-specific attributes—i.e `className` or `tabIndex` instead of `class` and `tabindex`, respectively—this is also the case for non DOM attributes and CSS classes defined in the project. WebStorm also suggests events that can be triggered on a specific element—i.e `onClick` or `onMouseOver` on a `div`-element. Self-made components are also suggested, thus the developer can easily reuse components—WebStorm also suggests the properties of these self-made components.

WebStorm comes preinstalled with a collection of code snippets[4] for React. Emmet[5] with self-made components is another feature that WebStorm supports.

Navigation in a React project is also supported by WebStorm. Everywhere a component is in use a quick view of the component is possible, and it is possible to navigate to the component to see the full definition. The JSX tree structure can at times be hard to navigate; WebStorm color codes the tree to make it easier to navigate.

WebStorm comes with built-in code inspections that inform about potential bugs, code style, and abnormalities in the code. An example use can be seen below in Fig-

[4]Small reusable statements or blocks of code that help developer be more productive.
[5]`https://docs.emmet.io`.

ure 6.4. WebStorm comes with a collection of quick fixes for common inspection errors—i.e. adding a missing method.

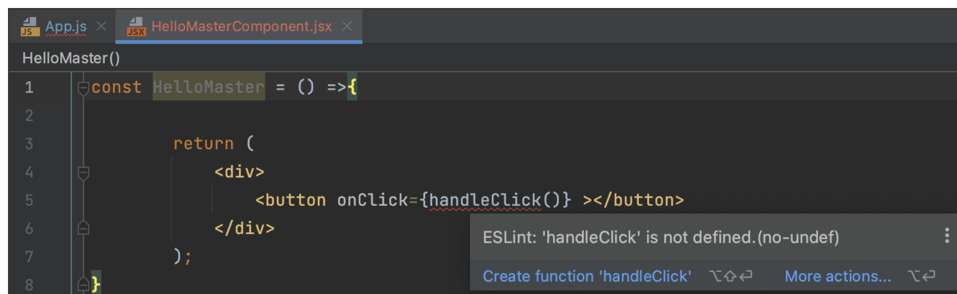


Figure 6.4: Inspection error showing a missing method, with a quick fix in WebStorm.

WebStorm allows for refactoring of code, such as renaming, converting and destructuring. Renaming of a component, method, state, etc. is carried out over the whole project, and not just in the current file. WebStorm allows for converting a functional component to a class component, and vice versa. Destructuring helps in unpacking a object into a set of variables.

Tests are a huge part of every industrial level application, and WebStorm has support for testing React applications with Jest[6].

6.2.2 Vue.js in VScode

Vue.js [13], often called Vue, is a JavaScript framework for building UIs. The framework has taken design inspiration from MVVM, and was developed as an open source project. Like React, Vue is a component based framework.

VSCode does not come with default support for Vue; in order to get support for Vue one must install one of the many plugins available on the VSCode marketplace. One of the most popular plugins is *Vetur*[7], which gives VSCode features like syntax highlighting, semantic highlighting, snippets, error checking, formatting, and so on. Syntax and semantic highlighting is a feature that colors the syntax of the code. Snippets allow for a collection of reusable code to be inserted into the code. Error checking is a feature that checks the code for errors. Formatting is a feature that formats the code to a specific style.

[6] https://jestjs.io
[7] https://marketplace.visualstudio.com/items?itemName=octref.vetur

6.2.3 Ktor in IntelliJ IDEA Ultimate

IntelliJ IDEA Ultimate comes with first-hand support for Ktor. It provides the functionality to create a Ktor project from scratch, right from within the IDE. When running a Ktor project, the IDE will autocomplete with Ktor specific plugins for our project. IntelliJ is aware about the API path the developer has created, and will give the developer autocomplete paths and sub-paths that correspond to the project paths. In a Ktor project, refactoring a API path can be tedious, as the developer must manually change the path in all the places it is used. Therefore, when running a Ktor project, the developer can refactor the path in one place, and the IDE will refactor the path in all the places it is used [7].

6.2.4 Vaadin in IntelliJ IDEA Ultimate

Vaadin [11] is an open source web application platform for Java development. Vaadin uses Java attributes to implement functionality. E.g., a statement `button.setText("Click me");` specifies that the caption of a button widget is “Click me”. Previous versions of IntelliJ Ultimate provided extensive support for the Vaadin framework: the IDE notified the developer if there where attributes that where given a value twice, or if a value was missing for an element to be placed on the canvas. Thus, the IDE was aware of the semantics of Vaadin.

IntelliJ IDEA Ultimate at the time of writing does not come with support for Vaadin. Vaadin Designer is a plugin for IntelliJ IDEA that works as a GUI designer for Vaadin. This plugin lets one graphically add components, drag and drop to a layout, and these changes are reflected in the Java code. The GUI also allows for assigning values to attributes, such as the text of a button, fixing the height and width of an element, and defining CSS attributes.

6.3 Tools for debugging web applications

Most developers use web development tools for debugging their web applications. These tools are often built as extensions for web browsers, and have to be written for a specific library or framework. These tools are used to make debugging of web applications easier, to make it easier to inspect the state of the application, and to map out the performance of the application. In this section we look at some of the web development tools that are used in web development.

6.3.1 React Developer Tools

React Developer Tools [23] is an open-source browser extension (Google Chrome, Mozilla Firefox and Microsoft Edge) that comes with a set of features that helps the developer debug their application. The extension adds two new tabs in the Chrome DevTools interface; *Components* and *Profiler*, which can be used when debugging a React application.

The tab *Components*, is a tree view of the React component tree. This tree contains the components rendered in the application, together with the props and state of the components. A developer is able to manipulate the state and props of the components, and see the changes in the application. The tab *Profiler* is a tool that helps the developer see the performance of the application and find bottlenecks in the application. There, a developer can see a *flame chart* that shows the time spent rendering different components.

6.3.2 Redux DevTools

Redux DevTools [18] is an extension for the browser (Google Chrome and Firefox). The extension allows the developer to follow the state as it changes over time—timeline debugging. The extension allows for manipulation of the timeline; removing an action, toggling an action, etc. The developer can choose between four different views of the timeline: the log monitor, chart monitor, diff monitor and inspector monitor. All of these different monitors are tailored to specific use cases.

6.3.3 Angular DevTools

Angular DevTools [1] is an extension for browser (Google Chrome and Firefox). The extension features a toolset that helps the developer debug their application. The extension adds a new tab in the Chrome DevTools interface: *Angular*. When this tab is open the developer can choose between two sub-tabs: *Components* and *Profiler*.

The *Components* tab describes the tree structure of the root component and subcomponents rendered in the application, together with the state and props of these components. The developer can interact with the components in the tree, and change the value of the state and props, and see the result in the browser. The *Profiler* tab allows the developer to get a better understanding of the execution of the application. The

developer can start a recording of the application, while recording the DevTools listens to change and hook executions. When the recording is done, the developer can see three different visualizations of the execution—*flame chart*, *tree map* or *bar chart*—and view details about the execution, such as execution time, parent hierarchy, etc.

6.4 Language workbenches

In modern software development, it is crucial to have a state-of-the-art IDE support, with support for code navigation, autocompletion, and refactoring capabilities, which should work even in erroneous states.

Tools to implement IDEs tailored for particular languages are called *language workbenches* [27, 16]. Fowler defines a language workbench as a set of tools where the user can define new languages that can be integrated with each other. Language workbenches should support the following three mandatory features for the language definition: *notation*, *semantics*, and an *editor*, but it may also have support for *validation* of models, *testing*, and *composability* of languages [24].

6.4.1 Xtext

Xtext is a framework from Eclipse [14] for development of DSLs. Xtext uses the compiler construction patterns and comes with common features, such as reusable expression language and workspace indexer [20]. Xtext inlcudes the Eclipse editor, which includes all of the capabilities outlines in Subsection 6.1.4, making it a highly sophisticated language workbench. From an EBNF grammar specification [19]—similarly to ANother Tool for Language Recognition (ANTLR) [17]—Xtext provide the AST, the lexer, the parser, type checker and a compiler for the grammar provided. Xtext supports the Language Server Protocol [14], and therefor: Xtext can be used with any IDE that supports the Language Server Protocol. There is also a possibility to embed the DSL editor in a web browser [17]. The use cases of Xtext range from small hobby projects to large scale production projects. This is because of the architecture that reuses well-established and commonly understood semantics for language aspects. Every aspect of the programming language implementation can be customized with dependency injections [24].

6.4.2 Spoofax

Spoofax is a language workbench that allows defining all language aspects using declarative DSLs [32]. Each such DSL offers a unique technique for reuse and modularity. The abstract and concrete syntax modules, for example, can import other modules for reuse or separation of concerns. Rules for rewriting are used in the dynamic semantics of a language. These are not pure rewriting rules for modularity reasons, but they have been supplemented with dynamic rules and programmable rewriting techniques. Although the ability to change the sequence of rewriting rules using strategies provides flexibility, the creation of such strategies may be time-consuming [39]. Spoofax uses Java classes to delegate modularity to the DSL, through inheritance and overriding [43].

6.4.3 Rascal

Rascal is meant for analysis and transformation of source code. Therefore Rascal is a *metaprogramming language* [41]. Rascal comes with integration with Eclipse, thus features like syntax highlighting, Read-Eval-Print-Loop (REPL), etc, are available for developers when using Rascal in Eclipse. These features can also be dynamically extended to DSLs. Rascal features a number of domain specific constructs, such as regular expressions, comprehensions, transitive closures, and it has most of the features of a GPL. The syntax of Rascal is familiar to curly based GPLs, such as C, Java, Kotlin, JavaScript or C#. Rascal is designed to support both large complicated tasks and small noncomplicated tasks. The first one could be task such as analyzing and transforming COBOL code (large number of lines), thus Rascal requires a powerful parsing technology to be able to handle the complicated grammar of COBOL and to be able to handle such volume of code. Rascal also lets the developer decide what to use for the analyzing and transforming the code instead of automatically helping the developer behind the scene—the developer can decide on the precision-performance tradeoff.

6.5 Visual specification

6.5.1 LabVIEW

LabVIEW, Laboratory Virtual Instrument Engineering Workbench, is a visual programming environment from National Instruments [33]. It is a tool that is developed for non-programmers, such as scientists and engineers, to create applications for measurement

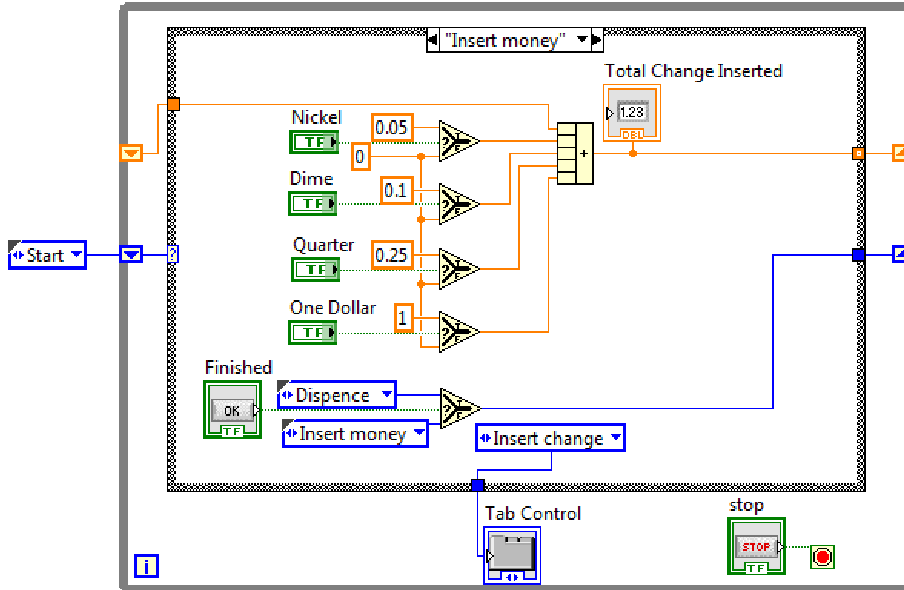


Figure 6.5: “LabVIEW State Machine example” by Smith Pohl, used under CC BY.

and testing. LabVIEW uses a graphical programming language, called *G* for Graphical programming language, which is describes as a ”graphical, structured dataflow programming language” [33]. The language consist of *blocks* and *wires*. A block is a graphical representation of a *structure node*—the equivalent to a function in traditional text based programming languages—and wires determine the flow of data between blocks. A structure node will not execute before all the inputs are available. Once a structure node has executed, it outputs the result down the wire to nodes further down the dataflow. The blocks and wires together form a 2-dimensional graph diagram.

The LabVIEW application uses a drag-and-drop interface, thus the user can drag-and-drop a block from the palette and drop it onto a canvas. On the canvas the user can connect the block to other blocks with wires, and repeat this process until the application is complete. From this interface the user can also run the application to test their implementation.

6.5.2 Graphviz

Graphviz is a set of tools for rendering graphs, from textural specifications in the DOT language. It contains different batch layouts: an incremental layout, graphical editors, a web server for including graphs in the browser, etc [21, 22]. Graphviz can be used as a command line tool, a library for the C programming language, a GUI, and via a web

browser [22, 21]. An important use case of Graphviz is to create graphs to be presented in publications, such as papers, reports, etc.

Chapter 7

Discussion and future work

In this thesis we have implemented tool support for specifying multi-way dataflow constraint systems. Our implementation features IDE support for the constraint system library HotDrink, in the form of an extension to VSCode. The extension provides syntax highlighting, code completion, error reporting for HotDrink constraint systems, as well as code generation and visualization of specifications written in the language. Code for the JavaScript Hotdrink API can be generated from the constraint system specification, a demo application can be provided from the specification, DSL code for visualization of relations between variables in a online graph generator, and a visualization of the constraint system as a graph.

Our implementation also features an extension to Google Chrome. This extension provides an overview of all HotDrink variables on a web page, and it can be used to inspect the state of a HotDrink constraint system. Currently in the VSCode extension, we generate a demo application in HTML and JavaScript; we could also generate a demo application for major frameworks, like React, Vue, Angular, etc., and also make it interoperability with other frameworks. This would allow for a more realistic demo application for a given developer, and a developer could easily see how a constraint system could be useful in their application. Another feature we could add is a DSL for specifying the IDE behavior, which would allow for more customization of the IDE, and ensure that future versions of HotDrink could be supported (without reimplementing the extension). We could, for example, add support for WarmDrink [38], an extension of HotDrink.

In the Chrome extension, a possible feature is a graph view of the constraint system history. With this the developer could see how the constraint system alters over time.

A natural following feature to this history graph is to be able to manipulate this history graph and see the effect throughout the constraint system history. The Chrome extension features a one-way communication channel between the web page and the extension. If this were to be a two way communication channel the extension could be used to alter the value of variables on the webpage.

To further evaluate our implementation, a large-scale usability study could be conducted. This would allow us to see how developers react to our extension, and how they use it.

List of Acronyms and Abbreviations

ANTLR ANother Tool for Language Recognition.

API Application Programming Interface.

AST Abstract Syntax Tree.

CSS Cascading Style Sheets.

DOM Document Object Model.

DSL Domain Specific Language.

EBNF extended Backus-Naur form.

GPL General Purpose Languages.

GUI Graphical user interface.

IDE Integrated development envirement.

LabVIEW Laboratory Virtual Instrument Engineering Workbench.

Less Leaner Style Sheets.

LSP Language Server Protocol.

MVVM Model-View-ViewModel.

PaaS Platform as a Service.

REPL Read-Eval-Print-Loop.

Sass Syntactically Awesome Style Sheets.

UI User Interface.

VCS Version Control System.

VSCode Visual Studio Code.

Bibliography

[1] Angular DevTools Overview.
URL: `https://angular.io/guide/devtools`. [Accessed on *2022-11-09*].

[2] Eclipse IDE.
URL: `https://www.eclipse.org`. [Accessed on *2022-6-10*].

[3] Google Chrome API.
URL: `https://developer.chrome.com/docs/extensions/reference/`. [Accessed on *2022-11-13*].

[4] Atom.
URL: `https://atom.io/`. [Accessed on *2022-07-25*]. A hackable text editor for the 21st Century.

[5] AWS Cloud9.
URL: `https://aws.amazon.com/cloud9/`. [Accessed on *2022-10-18*]. A cloud IDE for writing, running, and debugging code.

[6] Electron.
URL: `https://electronjs.org`. [Accessed on *2022-07-25*]. Build cross-platform desktop apps with JavaScript, HTML, and CSS.

[7] Ktor: IntelliJ IDEA Ultimate Support.
URL: `https://ktor.io/idea/`. [Accessed on *2022-11-15*]. IntelliJ IDEA Ultimate comes with first-class support for Ktor.

[8] Langium.
URL: `https://langium.org`. [Accessed on *2022-09-07*].

[9] PyCharm.
URL: `https://jetbrains.com/pycharm/`. [Accessed on *2022-08-13*]. The Python IDE for Professional Developers.

[10] React.
URL: `https://reactjs.org`. [Accessed on *2022-08-27*]. A JavaScript library for building user interfaces.

[11] Vaadin.
URL: `https://vaadin.com`. [Accessed on *2022-09-23*]. The modern web application platform for Java.

[12] Visual Studio Code.
URL: `https://code.visualstudio.com`. [Accessed on *2022-11-10*]. Code editing. Redefined.

[13] Vue.
URL: `https://vuejs.org`. [Accessed on *2022-09-23*]. The Progressive JavaScript Framework.

[14] Xtext.
URL: `https://www.eclipse.org/Xtext/`. [Accessed on *2022-07-12*]. Language Engineering Made Easy.

[15] Eclipse Sprotty™.
URL: `https://projects.eclipse.org/projects/ecd.sprotty`. [Accessed on *2022-09-07*].

[16] Mikhail Barash. Vision: the next 700 language workbenches. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering*, pages 16–21, 2021.

[17] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[18] Daniel Bugl. *Learning Redux*. Packt Publishing Ltd, 2017.

[19] Sven Efftinge and Markus Völter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, 2006.

[20] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. Xbase: implementing domain-specific languages for Java. *ACM SIGPLAN Notices*, 48(3):112–121, 2012.

[21] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz—open source graph drawing tools. In *International Symposium on Graph Drawing*, pages 483–484. Springer, 2001.

[22] John Ellson, Emden R Gansner, Eleftherios Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz and dynagraph—static and dynamic graph drawing tools. In *Graph drawing software*, pages 127–148. Springer, 2004.

[23] Elad Elrom. Debug and profile your React app. In *React and Libraries*, pages 337–370. Springer, 2021.

[24] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. The state of the art in language workbenches. In *International Conference on Software Language Engineering*, pages 197–217. Springer, 2013.

[25] Moritz Eysholdt and Johannes Rupprecht. Migrating a large modeling environment from XML/UML to Xtext/GMF. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 97–104, 2010.

[26] Gabriel Foust, Jaakko Järvi, and Sean Parent. Generating reactive programs for graphical user interfaces from multi-way dataflow constraint systems. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 121–130, 2015.

[27] Martin Fowler. Language workbenches: The killer-app for domain specific languages.
URL: `https://martinfowler.com/articles/languageWorkbench.html`. [Accessed on *2022-07-08*].

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

[29] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Commun. ACM*, 33(1):54–63, jan 1990. ISSN 0001-0782. doi: 10.1145/76372.77531.
URL: `https://doi.org/10.1145/76372.77531`.

[30] John Gossman. Introduction to model/view/viewmodel pattern for building WPF apps. October 2005.
URL: `https://learn.microsoft.com/en-us/archive/blogs/johngossman/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps`.

[31] Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob Smith. Algorithms for user interfaces. *SIGPLAN Not.*, 45(2):147–156, oct 2009. ISSN 0362-1340. doi: 10.1145/1837852.1621630.
URL: `https://doi.org/10.1145/1837852.1621630`.

[32] Lennart CL Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 444–463, 2010.

[33] Jeffrey Kodosky. LabVIEW. *Proceedings of the ACM on Programming Languages*, 4(HOPL):1–54, 2020.

[34] Lei Liu, Xinwen Zhang, Guanhua Yan, Songqing Chen, et al. Chrome extensions: Threat analysis and countermeasures. In *NDSS*, 2012.

[35] Jonas Kjær Rask, Frederik Palludan Madsen, Nick Battle, Hugo Daniel Macedo, and Peter Gorm Larsen. The specification language server protocol: A proposal for standardised LSP extensions. *arXiv preprint arXiv:2108.02961*, 2021.

[36] Michael Sannella. SkyBlue: A multi-way local propagation constraint solver for user interface construction. In *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology*, UIST '94, page 137–146, New York, NY, USA, 1994. Association for Computing Machinery. ISBN 0897916573. doi: 10.1145/192426.192485.
URL: `https://doi.org/10.1145/192426.192485`.

[37] Michael Sannella. The SkyBlue constraint solver and its applications. In *Proceedings of the 1993 Workshop on Principles and Practice of Constraint Programming*, pages 385–406. MIT Press, 1995.

[38] Knut Anders Stokke, Mikhail Barash, and Jaakko Järvi. A domain-specific language for structure manipulation in constraint system-based GUIs. *Journal of Computer Languages*. Has appeared.

[39] A.M. Sutii. *Modularity and reuse of domain-specific languages: an exploration with MetaMod*. PhD thesis, Mathematics and Computer Science, November 2017. Proefschrift.

[40] Rudi Blaha Svartveit. Multithreaded multiway constraint systems with Rust and WebAssembly. Master's thesis, The University of Bergen, 2021.

[41] Tijs van der Storm. The Rascal language workbench. *CWI. Software Engineering [SEN]*, 13:14, 2011.

[42] Brad Vander Zanden. An incremental algorithm for satisfying hierarchies of multiway dataflow constraints. *ACM Trans. Program. Lang. Syst.*, 18(1):30–72, jan 1996. ISSN 0164-0925. doi: 10.1145/225540.225543.
URL: `https://doi.org/10.1145/225540.225543`.

[43] M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L.C.L. Kats, E. Visser, and G.H. Wachsmuth. *DSL Engineering—Designing, implementing and using domain-specific languages*. M. Volter / DSLBook.org, 2013.

Appendix A

Grammar for the HotDrink DSL used in the implementation of the VSCode extension

Listing A.1: Langium grammar code for HotDrink DSL.

```
grammar HotDrinkDsl

entry Model:
    (imports+=Import)*
    (components+=Component)*
    ;

Import:
    "import" "{" imports+=ImportedFunction (","
      ↪ imports+=ImportedFunction )* "}" "from" file=STRING;

ImportedFunction:
    function=FuncName ("as" altName=FuncName)?
    ;
FuncName:
    name=ID;

Component:
    "component" name=ID "{"
        (variables += Vars |
        constraints += Constraint)*
    "}"
;

Vars:
    "var" vars+=Variable
    ("," vars+=Variable)* ";"
;

Variable:
    name=ID (":" type=VarType)? ("=" initValue=ValueExpr)?;

VarType returns string:
    'string'
    |
    'number'
    |
    'boolean'
    ;

ValueExpr:
    {infer StringValueExpr} val=STRING |
    {infer NumberValueExpr} (negative?="-")? digit=INT
      ↪ ("."decimal=INT)? |
```

```
43     {infer BooleanValueExpr} val=("true"|"false");
44
45 Constraint:
46     "constraint" (name=ID)?"{"
47         methods+=Method+
48     "}"
49 ;
50
51 FunctionCall:
52     funcRef=[FuncName] "(" (args+=[Variable] ("," args+=[Variable])*
        ↪ )?  ")"
53 ;
54
55 Method:
56     (name=ID)? signature=Signature "=>"
57         body=Body
58     ";"
59 ;
60
61 Body:
62     value=FunctionCall | value=Expr | "[" values+=Body (","
        ↪ values+=Body)+ "]";
63
64 VariableReference:
65     ref=[Variable] (hasMark?="!")?;
66
67 Signature:
68     "(" inputVariables+=VariableReference (","
        ↪ inputVariables+=VariableReference)* "->"
        ↪ outputVariables+=VariableReference (","
        ↪ outputVariables+=VariableReference)* ")";
69
70 Atomic infers Expr:
71     {infer NumberConst} (negative?="-")? digit=INT ("."decimal=INT)? |
72     {infer StringConst} value=STRING |
73     {infer BoolConst} value=("true" | "false") |
74     {infer VarRef} value=[Variable]
75 ;
76
77 Primary infers Expr:
78     {infer Parenthesis} "("expression=Expr")" |
79     {infer Not} "!" expression=Primary |
80     Atomic
81 ;
82
83 MulOrDiv infers Expr:
84     Primary ({infer MulOrDiv.left=current} op=("*"|"/")
        ↪ right=Primary)*;
85
86 PlusOrMinus infers Expr:
87     MulOrDiv ({infer PlusOrMinus.left=current} op=("+"|"-")
        ↪ right=MulOrDiv)*;
88
89 Comparison infers Expr:
90     PlusOrMinus ({infer Comparison.left=current}
        ↪ op=(">="|"<="|">"|"<") right=PlusOrMinus)*;
91
92 Equality infers Expr:
93     Comparison ({infer Equality.left=current} op=("=="|"!=" |"==="|
        ↪ "!==") right=Comparison)*;
94
95 And infers Expr:
96     Equality ({infer And.left=current} op=("&&") right=Equality)*;
97
98 Or infers Expr:
99     And ({infer Or.left=current} op=("||") right=And)*;
100
101 Expr: Or;
```

```
102
103 hidden terminal WS: /\s+/;
104 hidden terminal ML_COMMENT: /\/\*[\s\S]*?\*\//;
105 hidden terminal SL_COMMENT: /\/\/[^\n\r]*/;
106
107 terminal ID: /[_a-zA-Z][\w_]*/;
108 terminal INT returns number: /[0-9]+/;
109 terminal STRING: /"[^"]*"|'[^']*'/;
```

Appendix B

Binder functions generated by the demo application feature

In the code below the binder function takes three arguments, the first, `element`, is the HTML element that should be bound to the HotDrink variable, the second, `variable`, is the HotDrink variable that should be bound to the HTML element, and the third, `type`, is the HTML element attribute type the value should be bound to.

Listing B.1: Binder functions for binding HTML elements and HotDrink variables, generated be the demo application feature.

```
function binder(element, value, type) {
    value.value.subscribe({
        next: val => {
            if (val.hasOwnProperty('value')) {
                element[type] = val.value;
            }
        }
    });
    element.addEventListener('input', () => {
        value.value.set(element[type]);
    });
}

export function stringBinder(element, value) {
    binder(element, value, "value");
}

export function numberBinder(element, value) {
    binder(element, value, "valueAsNumber");
}

export function checkedBinder(element, value) {
    binder(element, value, "checked");
}
```

Appendix C

Coe validations supported by HDCode

Name	Description	Severity	Applicable to	Quick fix
`hintToInitialize-VariablesToZero`	If all variables are missing an initial value (undefined), this rule is activated.	Hint	Variable declaration	⊠
`checkVarStartsWith-Lowercase`	Adds warning underlining the variable name, if a variable name starts with a capital letter.	Warning	Variable	⊠
`checkSignatureOnly-ReferenceToVarOnce`	If the method tries to use a variable more than once in the signature.	Error	Method signature	☐
`checkSignatureFor-ExclamationVariables`	If the method signature uses an exclamation mark on one of the variables in the signature.	Info	Method signature	☐
`checkMethodStartsWith-Lowercase`	Adds warning underlining to the method, if a method name starts with a capital letter.	Warning	Method	☐

checkMethodBody-ReturnsToRightNumber-OfVariables	Checks if the return statement in the method body returns a list with the same length as the number of return variabels in the method signatur.	Error	Method	☐

checkConstraintStart-WithLowercase	Adds warning underlining to the constraint, if a constraint name starts with a capital letter.	Warning	constraint	☐
checkConstraint-MethodsHaveUniqueName	Checks that all method names in the constraint have unique names.	Warning	Constraint	☐
checkConstraint-MethodsUsesTheSameVars	Checks that all methods inside an constraint uses the same variables in their signatures.	Error	Constraint	☐
hintToMakePermutations	Checks that the constraint have only one method in it.	Hint	Constraint	⊠
hintToRemoveConstraint	Adds a hint to all constraints.	Hint	Constraint	⊠

checkComponent-ConstraintsHaveUnique-Name	Checks that all constraints inside an component has unique names.	Warning	Component	☐
checkComponentVars-HaveUniqueName	Checks that all variables in side the component has unique names.	Error	Component	☐
checkComponentFor-UnusedVariables	Adds a warning to all variables that are not used by any of the constraints in the component.	Warning	Component	☐

checkModelImpFunction-IsntImportedMoreThen-OnceInOnceStatement	Checks that a function is not imported more than once in an import statement.	Error	Model	☐

checkModelComponent-NameIsUnique	Checks that all components in the model has unique names.	Warning	Model	☐
checkModelForUnique-MethodNames	Checks that all methods in a model have unique names.	Warning	Model	☐

Table C.1: All validation rules implemented in HDCode.