University of Bergen
Department of informatics

Declaratively Programming the Dynamic Structure of Graphical User Interfaces

Author: Knut Anders Stokke
Supervisor: Jaakko Järvi

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

May, 2020

Abstract

Structures are everywhere around us—from chemical formulas to biological systems or musical works. In this thesis, we focus on composite structures that appear in graphical user interfaces (GUI), such as lists, tables, or tabs. GUIs often support changes to these structures—be it rearranging elements or appending new ones—in order to facilitate a more productive interaction between the user and the software system.

In general, making structural changes that involve GUI components is non-trivial: because the program state is stored and represented both in an object model and view widgets, structural changes to either representation should be reflected in the other. Furthermore, components in GUIs can be connected, and these connections must be updated whenever the structure of the components change. Ultimately, the intent of the operation, a structural change, is lost in the clutter of imperative statements that update the view, model, and connections between the components.

In this thesis, we define a framework that lets programmers specify possible structural changes in programs declaratively. We show how programmers can put these declarative specifications to use in JavaScript applications using a custom DSL we have implemented for the framework. To test our framework and demonstrate the clarity that it can bring to managing the dynamic structure of GUIs, we implemented an event scheduling application based on a library for defining and solving multi-way dataflow constraint systems in GUIs. The application has a sequence of timed events, and each event has a start time, duration, and end time. Whenever the user updates an event's duration, the start and end time of all the events are updated accordingly. The application supports structural changes, such as adding, removing, and reordering of events. We explain how components in these GUIs are connected and how we utilize our framework to correctly update the connections whenever structural changes are made.

Acknowledgements

I would first like to thank my supervisor Prof. Jaakko Järvi. Our research has benefitted from his extensive programming knowledge, especially in the field of GUI programming. His encouragement, patience, and constructive criticism throughout my time of research and writing have been invaluable for this thesis.

Secondly, I'm deeply indebted to Dr. Mikhail Barash for the interest he showed our research while I visited Prof. Järvi in Finland, and for the countless video calls since. The accomplishment of this thesis would not have been possible without his help and his expertise in domain-specific languages.

I am also grateful to Assoc. Prof. Anya Helene Bagge for constructive criticism of the thesis.

Finally, I would like to extend my sincere thanks to my wife and my parents for their support throughout my studies.

Knut Anders Stokke
18 May, 2020

Contents

1 Introduction **1**

2 Related Work **5**

3 Declarative Specification of Structural Changes **9**

3.1 Definitions 9

3.1.1 Defining relation specifications 9

3.1.2 Example of a relation specification 10

3.1.3 Defining rules 11

3.1.4 Adjacent components in lists 11

3.2 A DSL for structural changes 12

3.2.1 Components 12

3.2.2 Placeholders 13

3.2.3 Relation specifications 14

3.2.4 Rules 15

3.2.5 Swapping adjacent components in linked lists 15

3.2.6 Simultaneously swapping in two lists 16

3.2.7 Inserting between adjacent components 17

3.2.8 Swapping elements in a linked list 17

3.2.9 Inserting into and removing from a container 18

3.2.10 ApplyTexas 19

3.3 Summary 24

4 Implementation **25**

4.1 Eclipse and Xtext 25

4.2 Code generation 26

4.2.1 Specification of procedures 27

4.2.2 Specification of JavaScript code blocks 28

4.2.3 Specification of components 29

4.2.4 Specification of placeholders 30
4.2.5 Specification of relations 31
4.2.6 Specifications of transformation rules 34

5 Structural changes in constraint system-powered GUIs 39
5.1 Multi-way dataflow constraint systems 39
5.2 HotDrink 42
5.3 Example 46
5.3.1 The scheduler 46
5.3.2 Setup constraint system using HotDrink 47
5.3.3 Implementing structural changes with WarmDrink 48
5.4 Summary 52

6 Discussion and future work 53

Bibliography 57

List of Figures

3.1 Swapping adjacent components in linked lists. 16
3.2 Form for extracurricular acitivities in the ApplyTexas application. 20
3.3 Structural changes in the ApplyTexas GUI: specification of components, placeholders and relations. 22
3.4 Structural changes in the ApplyTexas GUI: specification of transformation rules. 23

4.1 Transpiling components to JavaScript. 29
4.2 The generated JavaScript function for establishing the relation `myRel`. . . 33
4.3 The generated JavaScript function for unestablishing the relation `myRel`. 33
4.4 The JavaScript function generated from the rule `myRule`. 36
4.5 Transpiling rules to JavaScript. 37

5.1 Example GUI form for rectangle measurements. 40
5.2 Connected HotDrink components. 45
5.3 The GUI of the event scheduler example. 46
5.4 Specification of an event component in the HotDrink DSL. 48
5.5 The relation specification `precedes`. 50
5.6 Transformation rules in the event scheduler. 51

Chapter 1

Introduction

Structures are everywhere around us — from chemical formulas, biological systems, musical works to graphical user interfaces. Some structures can be altered without altering the items of the structure: for instance, in a file explorer showing all the files and folders in a specific folder, we can move files from one folder to another. This changes the structure of the file system, but leaves the files unaltered.

In programming, we often find representations of data where we can make a distinction between *elements*, atomic data items, and the *structure*, how the elements are organized. By a *structural change* we refer to a change in the latter aspect, a change in how elements are organized. Examples of structural changes are adding elements to a structure, removing elements from a structure, swapping elements in a structure, and moving an element to a specific position in the structure. A structural change leaves the internals of the elements in a structure unchanged; only the relations between elements in the structure are changed.

In many cases, structural changes are easy to implement. For instance, adding elements to and removing elements from a list usually requires just one statement. Swapping two elements in a list might require three statements. In these examples, the structure is a carefully designed data structure—the list. Often the structures a programmer encounters are not carefully designed, but are instead quite ad-hoc. Such structures are commonly found, often as lists or grids, in graphical user interfaces (GUI). For example, multi-city reservations in a flight booking application have several flight segments; the segments are *listed*, with each segment displaying the departure and arrival information. In a flexible GUI, the customer would be able to remove and insert segments anywhere

in the list. Another example is that of task organization applications, which let users manage tasks in lists, and move tasks between the lists. A typical alarm application is yet another example: users add, remove and edit alarms, and the list of alarms should be automatically sorted on the alarm time. A further example is that of invoice generators in financial applications, where an invoice is a list of items and each item consists of a unit price, VAT, quantity, and a total price. Again, similarly to the previous examples, the user expects that items can be rearranged. The GUIs in all these examples have *dynamic* structures: elements can be added, moved, and removed on user interaction.

Making changes to ad-hoc GUI structures is often involved: data exists in both a model and a view and, thus, if the programmer changes the structure of either the model or the view, the structure of the other must also change. Elements in the structure can be complex, and the operation of moving one element may consist of several smaller operations. Furthermore, there are often various bindings (e.g. registered event handlers) between components in GUIs, and when making a structural change in such GUIs, the programmer must also update the relevant bindings. Ultimately, the intent of the operation, a structural change, is lost in the clutter of statements that update the view, model, and bindings between components.

The goal of this thesis is to introduce an abstraction layer for declaratively specifying changes to the structure of complex representations of data, without manually managing the lower-level connections between the components in the structure. The declarative specifications of structural changes are expected to make code easy to read and maintain. We aim at designing and implementing a framework where the programmer first defines the relations between the components that form a structure, together with lower-level operations involved in establishing or unestablishing these relations, and then specifies the structural changes by declaring *how* the relations between the components change. To make the specifications of structural changes concise, we designed and implemented a DSL, *WarmDrink*, for our framework. The framework transpiles the DSL to JavaScript, providing the programmer with JavaScript functions for making structural changes effortlessly. The framework is independent of GUIs, but the application we have in mind is GUIs, or more precisely, constraint system based GUIs.

Dependencies between variables in GUIs are common: when a user changes one GUI element, others need to be updated too. *Multi-way dataflow constraint system*-powered GUIs [25, 18] enlist constraint systems to manage such dependencies. When developing constraint system-GUIs, the programmer defines a *property model* [17] for the GUI, a specification of the constraints between the variables of the GUI. These constraints

constitute a constraint system that is invalidated when variables change. The system is solved by updating the variables in a way that enforces all the constraints. Constraint systems can be solved by generic algorithms, making the tedious task of understanding the details of how a change in one variable affects other variables in a GUI unnecessary.

Whenever a variable in a constraint system-based GUI is updated, the mechanics of the GUI updates the dependencies automatically. It does so by *listening* to each of the variables so that whenever one is changed, the rest are immediately updated.

Setting up dependencies between variables in a GUI is messy when the structures of the GUI changes. GUIs are hierarchies of components, and variables of separate components can have dependencies on each other, creating connections between components. Whenever a component is inserted into, removed from, or moved inside a structure of connected components, the connections between the components must update accordingly.

We give an example to showcase how components are connected and why the connections must update on structural changes: consider a component A with a property x, and a component B with some number properties y and z; these are denoted as $A.x$, $B.y$, and $B.z$, respectively. The three properties have numerical values. We define a constraint on the properties: the value of $A.x$ is the product of $B.y$ and $B.z$. Because the constraint is defined on properties of both A and B, the constraint *connects* A and B. Assume that in addition to the components A and B we introduce a new component C with property x, similarly to A. Then, a structural change that replaces A with C in the GUI would involve disconnecting A from B and connecting C to B. In other words, the constraint on $A.x$, $B.y$ and $B.z$ would be replaced with the same constraint on $C.x$, $B.y$ and $B.z$. Our framework helps programmers to largely ignore setting up dependencies between properties of components when changing the structure of the components.

The structure of the thesis is as follows. In Chapter 2, we overview previous results that relate to specifications of structural changes. In Chapter 3, we sketch the ideas of a framework that lets programmers declare structural changes in programs, as well as discuss how programmers can adopt these ideas in JavaScript applications using a custom DSL we developed for the framework. We describe the implementation details of the framework and the DSL in Chapter 4. Next, in Chapter 5, we implement an event scheduling application based on a library for defining and solving multi-way dataflow constraint systems in GUIs. We explain how components in such GUIs are connected and how our framework can be utilized to correctly update the connections on changes to widgets in the GUIs. We discuss future work in Chapter 6.

Chapter 2

Related Work

Our work has connections with several areas of programming and programming research. Our goal is to separate the structure of data into an isolated concern and, thus, we can see a connection to aspect-oriented programming [21]. Our approach involves developing a (relatively simple) specification language, so we lean on DSL technologies. We primarily aim to make GUI programming easier and more manageable, so we ought to explain how our work relates to the latest GUI programming approaches. Our work, at a general level, is about describing quite arbitrary structures and modifications to them. Graphs and hypergraphs are what can be used in such representations and, therefore, our work is connected to topics like hypergraph grammars and graph algebras. In this section, we elaborate on these themes, and structure this section accordingly.

React

Plenty of JavaScript libraries for building user interfaces have emerged the last decade, and many of them share similar approaches to update GUIs: programmers make changes to the model of the GUI and the library takes care of updating the view accordingly. One such library is *React* [3]: React is *component*-based, where each component manages its own state and view, and the entire UI is a composition of components. More specifically, each component object is an instance of the class `Component`. `Component` has two fields: `state` holds the internal state of the component and `props` holds data, or `properties`, that is provided and managed by other components. To render a component, the class `Component` has a method `render()` that each component overrides to return an HTML DOM representation of the component.

Here is an example of the JavaScript `render()`-method in a simple component definition:

```
class HelloWorld extends React.Component {
  render() {
    return (
      <div>
      Hello World!
      </div>
    );
  }
}
```

The `HelloMessage` component's `render()`-method returns the DOM node `div` with the text content "Hello World!". The DOM representation returned by the `render()`-method of a component can contain data from the state and properties of that component. Whenever the state or properties of the component change, the component is re-rendered. Thus, the `render()`-function of a component is a projection from the component's state and properties.

Each component manages its own state; a component cannot change the state of other components. A React component can listen to interactive HTML components, such as text inputs and buttons, and update its state on user interaction. The programmer changes the state of a React component by using the component's `setState()`-method, which also triggers a re-render of that component.

React components are hierarchical: a component can contain child components that are rendered inside their parent component. Even though components cannot access other components, components can pass data down to their child components' `props`-fields. The data of a child component can, therefore, change due to a state change in one of the (transitive) parents of the component.

React reacts to state updates by updating the view. Making updates directly to the DOM in the browser is inefficient and React, therefore, uses a *virtual DOM* to update the view. A virtual DOM is a copy of the *real DOM* that the browser renders for the user to see. The virtual DOM is used to compute the minimum changes React has to make to the real DOM to re-render a component. A component needs to be re-rendered if its state has changed, or if its properties have changed due to a state change in a parent component. Thus, whenever the state of a component is updated, the component and all its transitive child components are re-rendered. React runs the render-method of the

component in which the state changed, and the method returns a virtual DOM that is the new DOM representation of that component and its children. It then computes the *minimum edit distance* from the old virtual DOM to the new virtual DOM. An edit is a deletion or insertion of a DOM node, and the minimum edit distance is the minimum number of edits needed to update the old virtual DOM to match the new virtual DOM. React finishes re-rerendering by performing these edits to the real DOM.

From a programmer's point of view, structural changes, such as reordering of list items, are easy in React: the programmer performs changes directly to the model, and the view is automatically updated. More specifically, the model in a React GUI is a composition of the state of all the components, and thus the programmer changes the model by changing the state of the components. Changing the state of a component triggers a re-rendering of that component, as described above. DOM nodes are removed and inserted until the component is rendered and the view matches the state of the components. When making these changes, the programmer only cares about the model, and React updates the view accordingly. Such simplicity, however, only manifests when the structure being modified is itself simple.

Structural changes in GUIs can involve components with *connections*, such as dependencies between variables of the components. For example, assume two components A and B whose states contain the numerical variables vA and vB, respectively. Assume further that vB is twice the value of vA. Because vB has a dependency on vA, the two components A and B are connected. If a structural change removed A and replaced it with another component C, we would have to tear down the connection between A and B and set up a similar connection between C and B. React, and other popular, component-based frameworks have no mechanisms for tearing down and setting up connections between components in this manner. Therefore, on structural changes that involve changes to component connections, the programmer must manage component connections manually.

Our abstraction enables the programmer to specify structural changes to both the model and the view of the GUI at a higher level, instead of having to make changes directly to the model and the view. When applying the specifications for structural changes in GUIs, the programmer should not have to manage connections between components; the specifications are an abstraction layer over component connections. In the example above, the programmer would use such a specification to replace A with C, without having to manually disconnect A from B and connect C to B.

Hypegraph Edge Replacement Grammar

A possible approach to specifying structural changes is using *Hypergraph Edge Replacement Grammar (HERG)* [10]. A *Hypergraph* is a generalization of a graph where the edges, which are named *hyperedges*, connect any number of vertices. Hypergraph edge replacement grammars define production rules for replacing hyperedges in hypergraphs with new hypergraphs and, by that, expand the hypergraph. The hyperedges that can be replaced with hypergraphs are called *expansion points*, and are similar to non-terminals in context-free grammars.

We considered using HERGs to specify structural changes in GUIs. We could describe GUIs as hypergraphs where the vertices are components and the hyperedges are connections between components. We could then specify insertions of components into the GUI with HERGs, by using expansion points in the GUI hypergraph where components were allowed to be inserted. By executing a production rule to the GUI hypergraph, a component would be inserted into the view and model of the GUI.

While HERGs enable us to generate and expand hypergraphs, HERGs are less suitable for removing (sub)hypergraphs from the hypergraph and exchanging subgraphs in the hypergraph. Thus, HERGs are not suitable for specifying some types of structural changes that are encountered in GUI programming, such as removals and reorderings of components. Still, in our approach, we draw inspiration from HERGs and the idea of representing components and their connections as hypergraphs.

Chapter 3

Declarative Specification of Structural Changes

As discussed above, imperatively performing structural changes in a GUI is messy. Each structural change involves adding, removing, or reorganizing GUI elements, and also unestablishing and establishing various relations between them — for example, event handlers may need to be removed or registered. Such relations are implicit in code, and the intention is easily lost in the clutter of statements.

We made these relations explicit with a framework for describing structural changes on component graphs in a declarative manner. In this framework, changes are described using *transformation rules* from one set of relations to another. Using *relation specifications* that specify how relations are established and unestablished, generic algorithms can perform structural changes by first unestablishing the relations that are to be unestablished and then establishing those that are to be established.

3.1 Definitions

3.1.1 Defining relation specifications

A *component* in our framework is any value associated with a type. It can be one JavaScript object, part of one object, or a composition of many objects. A *relation*

specification $\mathcal{R}$ is a 4-tuple $\langle V, r, e, u \rangle$, where V is a k-element sequence of typed component variables, r is a k-ary relation on V and e and u are procedures, each taking V as its parameter list. If r does not hold on V, then calling e with V will establish r on V. If r holds on V, then calling u with V will unestablish r on V. We say that $\mathcal{R}$ holds for some V iff r holds for that V.

The procedures e and u are optional, and either may be empty if establishing or unestablishing is not applicable for the relation. If both e and u are empty, however, the relation specification is of no use.

3.1.2 Example of a relation specification

In the next example, we define a relation specification for a relation between *HTML Document Object Model (DOM)* nodes. The DOM-model is a tree data structure where each node can have a finite sequence of child nodes. Each node has a pointer, `nextElementSibling`, to the next element in the sequence. This pointer is `null` if the element it belongs to is the last element in the sequence. Each node also has a pointer, `parentElement`, to its parent node. In the following example of a relation specification we use JavaScript to act on an HTML DOM.

We define the relation *precedes* for two DOM-nodes $a, b \in C$: $\langle a, b \rangle \in$ *precedes* iff a and b are in the same sequence and b directly follows a. To establish this relation, we use the procedure *establishPrecedes*. It is an ordinary JavaScript arrow function that inserts b after a in the sequence containing a:

```
(a,b) => a.parentElement.insertBefore(b, a.nextElementSibling)
```

In this relation specification we choose to leave the unestablish procedure empty. If necessary, we could define it to remove `b` from the sequence that `b`'s parent maintains.

With the establishing procedure available, we can define the *precedesSpec* relation specification. We use square brackets to denote sequences.

$$precedesSpec = \langle [a, b], precedes, establishPrecedes, empty \rangle \tag{3.1}$$

The meaning of the relation specification is as follows. Assuming two components x and y whose types correspond to the types of a and b, respectively, one can establish the

relation *precedes* on x and y by executing the procedure *establishPrecedes* with x and y as the arguments.

As mentioned, the relation specification has no unestablish procedure to unestablish the relation on the two components if *precedes* is established on x and y. The relation is, however, unestablished as a side-effect of establishing *precedes* on x and another component or establishing *precedes* on another component and y; for instance, if one establishes *precedes* on x and another component z, z is moved in between x and y and thus, *precedes* no longer holds on x and y.

3.1.3 Defining rules

Denote by R the set of all relation specifications. Let $c_1, ..., c_n \in C$ be components. A *transformation rule* on $c_1, c_2, ..., c_n$ is a triple $\langle [c_1, c_2, ..., c_n], Pre, Post \rangle$ where $Pre \subseteq R$ is a conjunction of relation specifications that hold on $c_1, c_2, ..., c_n$ before the transformation and $Post \subseteq R$ is a conjunction of relation specifications that hold on $c_1, c_2, ..., c_n$ after the transformation. An application of a transformation rule unestablishes all the relations in *Pre* and establishes all the relations in *Post*. The relation specifications in *Pre* must have a non-empty unestablish procedure. Likewise, the relation specifications in *Post* must have a non-empty establish procedure.

Given a transformation rule and a list of components whose types respectively conform to the types of the arguments of the rule, a generic algorithm can perform the transformation of the rule by first unestablishing the relations in *Pre* and then establishing the relations in *Post*.

3.1.4 Adjacent components in lists

We give an example of a transformation rule in our framework. The example rule swaps adjacent nodes in a sequence of HTML DOM nodes. Using the relation *precedes* defined above, the rule to swap two adjacent HTML nodes a and b is very simple:

$$swap = \langle [a, b], \{\langle a, b \rangle \in precedes\}, \{\langle b, a \rangle \in precedes\} \rangle$$

The first component of *swap* states that *swap* is a rule defined for two components $a, b \in C$. The second part, *Pre*, states that a is expected to precede b before the transformation. The last part, *Post*, states that b should precede a after the transformation.

The rule *swap* is declared using only relations expected to hold before or after the transformation. We will discuss in Chapter 4 how a generic algorithm can apply this transformation on two given components a and b by first running the unestablish procedures of the relation specifications in *Pre* and then the establish procedures of the relation specifications in *Post*.

In our case, since the relation specification *precedesSpec* in *Pre* has an empty unestablish operation, no procedure is run from *Pre*. The procedure from *Post* that is run is the establish code in *precedesSpec*, which happens to be the only procedure needed to perform the transformation:

```
b.parentElement.insertBefore(a, b.nextElementSibling)
```

The point of this very simple example is merely to explain the structure of rules. Many of the relation specifications and transformations rules we define below will result in more procedures being run when the transformation rules are applied.

3.2 A DSL for structural changes

We now introduce a *domain-specific language* (DSL) for defining relation specifications and rules as code. The DSL is transpiled to the language the programmer uses to write GUI code. The DSL consists of five sections: the *code section*, *components section*, *placeholders section*, *relations section* and *rules section*. The code section is optional, it allows for arbitrary code to be written and copied into the generated code, which is useful for imports, constants, and helper functions.

3.2.1 Components

The *components section* is where *components' types* are defined. Each type t has a name and a procedure to test whether a component complies with t. The framework uses the procedure to type-check the component parameters involved in a structural change and

warns the programmer if the parameters do not conform to the expected component type. The procedure is wrapped in triple quotes (`'''`) and the component is referred to by using the name of the type in Guillemets (`«»`). Components themselves are constructed outside of the framework and are only manipulated by the framework when they are involved in a structural change.

Here are two examples of component type definitions in the DSL:

```
components
  Text <-> '''typeof «Text» === "string"'''
  Li <-> '''«Li» instanceof HTMLLiElement'''
```

The code defines the two component types `Text` and `Li` and the codes for testing compliance to these types. The test codes differ because JavaScript uses different means for inspecting the type of values (`typeof`) and objects (`instanceof`).

3.2.2 Placeholders

The *placeholders section* is where *placeholders* are defined. Placeholders are essentially typed variable names. They are used in the definitions of relation specifications and rules; they represent component arguments in parameter lists. Each placeholder has a name and a component type from the components section, as seen in this example:

```
placeholders
  t: Text
  a, b: Li
```

When using `t` in a relation specification, the specification expects its parameters to comply with the component type `Text`. Placeholders let the application programmer define relation specifications and rules without explicitly declaring the types of the parameters involved. The reason for having placeholders is further demonstrated in the following sections.

3.2.3 Relation specifications

The *relation specifications section* is where relation specifications are defined. Each relation specification has a name, a list of placeholders, a test procedure, an optional establish procedure and an optional unestablish procedure. The procedures are wrapped in triple quotes, and placeholders are referred to in the code by using their names in guillemets («»). As discussed above, either the establish procedure or unestablish procedure may be empty, but one of the two procedures has to be non-empty.

In the following example of a relation specification, we assume the class `LinkedListNode` to be defined, and that instances of the class are nodes in a linked list. We assume the instances to have the field `pointer` that points either to the next node in the list or has the value `null`.

```
components
  Node <-> '''«Node» instanceof LinkedListNode'''

placeholders
  x, y: Node

relations
  (pointsAt) x y ::=
    test '''«x».pointer === «y»'''
    establish '''«x».pointer = «y»'''
    unestablish '''«x».pointer = null'''
```

The `pointsAt` relation specification has three procedures. The first procedure, the test procedure, tests whether the relation holds on two given components `x` and `y` by checking if `x` points to `y`. This procedure is used to check that the relation, in fact, holds after it is established or does not hold after it is unestablished. This guides the application programmer to write correct relation specifications. The second procedure, the establish procedure, establishes the relation on `x` and `y` by referring `x.pointer` to `y`. The third and last procedure, the unestablish procedure, unestablishes the relation for two given components `x` and `y` by setting `x.pointer` to `null`, and thus splits the linked list.

Note that the conceptual relation, r, of the relation specification is not explicitly given. We could define the relation of the relation specification *pointsAt* to be the following:

$$\langle x, y \rangle \in \mathit{pointsToRel} \text{ iff the field } \texttt{pointer} \text{ of } x \text{ points to } y$$

This relation definition is not easily interpreted by a program. Therefore, instead of giving a relation definition to the framework, we codify it into the test procedure of the relation specification. The conceptual relation of any relation specification should be obvious from the test procedure as well as from the name of the specification.

3.2.4 Rules

The *rules section* is where *transformation rules* are defined. Each rule has a name, list of placeholders, conjunction of relations on the placeholders that should hold before the transformation, and conjunction of relations on the placeholders that will hold after the transformation. The relations in each list are separated by commas, and the two lists are separated by an arrow (`=>`).

Here is an example of a transformation rule:

```
rules
  flipPointer (x y) = x pointsAt y => y pointsAt x
```

The name of the transformation rule is `flipPointer`. Assume two components `a` and `b`, where the types of `a` and `b` correspond to the types of the placeholders `x` and `y`, respectively, and that `a` points at `b`. Applying the transformation rule `flipPointer` to `a` and `b` will unestablish the relation `a pointsAt b` and establish the relation `b pointsAt a` so that `b` points at `a` after the transformation.

3.2.5 Swapping adjacent components in linked lists

Using our DSL, we specify in Figure 3.1 the relation and rule for swapping adjacent elements in a linked list. The code specifies how to swap two adjacent components `x` and `y` in a linked list. Before the transformation, it expects the component `a` to precede `x`, `x` to precede `y`, and `y` to precede the component `b`. After the transformation, `a` should precede `y`, `y` should precede `x`, and `x` should precede `b`.

Figure 3.1: Swapping adjacent components in linked lists.

```
components
  Node <-> '''«Node» instanceof LinkedListNode'''

placeholders
  a, x, y, b: Node

relations
  (precedes) x y ::=
    test '''«x».pointer === «y»'''
    establish '''«x».pointer = «y»'''
    no unestablish

rules
  swap (a x y b) = a precedes x, x precedes y, y precedes b
    => a precedes y, y precedes x, x precedes b
```

3.2.6 Simultaneously swapping in two lists

The next example demonstrates how a structural change on several structures at once can be defined using the same transformation rule. This is beneficial in GUI-programming as changes occur to the model and view simultaneously. Furthermore, if the model is projected to several views, a change to the model be projected to all those views simultaneously.

Consider two linked lists of the same length that are element-wise related: if the order of the elements changes in one, a matching change should be made to the other. Swapping two adjacent elements in one of the lists therefore involves swapping the corresponding elements in the other list. To define such an operation we first define a component representing a pair of elements, one for each of the lists:

```
components:
  Pair <->
    ''' «Pair».fst instanceof LinkedListNode &&
        «Pair».snd instanceof LinkedListNode
    '''

placeholders:
  a, x, y, b: Pair
```

We then define a binary relation *precedesInBoth*, similar to the relation *precedes* in the previous example. The test procedure of *precedesInBoth*, however, checks that the relation holds for both of the lists; the establish code also acts on both lists.

```
(precedesInBoth) x y ::=
  test
    ''' «x».fst.pointer === «y».fst &&
        «x».snd.pointer === «y».snd '''
  establish
    ''' «x».fst.pointer = «y.fst»
        «x».snd.pointer = «y.snd»
    '''
  no unestablish
```

Furthermore, the rule to swap elements in both lists *swapInBoth* is declared as

```
swapInBoth (a x y b) =
  a precedesInBoth x, x precedesInBoth y, y precedesInBoth b
  => a precedesInBoth y, y precedesInBoth x, x precedesInBoth b
```

As rules are defined using relations, the definitions of the rules stay clean even for more complicated components and relations. The rule in this example is no more complicated than the rule *swap* from one of the previous examples.

3.2.7 Inserting between adjacent components

The next example defines a rule to insert a component between two adjacent components in a list. To insert a component x between two adjacent components a and b in a list, we keep the already defined relation *precedes*, and define a transformation rule *insertBetween*:

```
insertBetween (a x b) = a precedes b => a precedes x, x precedes b
```

3.2.8 Swapping elements in a linked list

Previously, we looked at swapping adjacent elements in linked lists. In this example, we look at swapping non-adjacent elements in a linked list. This is more involved, as we have to take into account the context. Swapping elements a and b in a linked list

involves making changes to the predecessor and successors of the two elements, and there are several cases: a and b can be adjacent or not, a can be before or after b in the list, a or b can be the first element in the list or not, and a or b can be the last element in the list or not.

The combination of all the cases would involve 32 rules and we, therefore, can introduce *nullable placeholders* in rules to the language by giving a question mark after the name of the placeholder. Nullable placeholders can be empty, and relations on `null`-placeholders are ignored. Such a feature would allow us to define more rules in one rule definition:

```
swapLinked (a? b c? x? y z?) =
  a precedes b, b precedes c, x precedes y, y precedes z
  =>
  a precedes y, y precedes c, x precedes b, b precedes z
```

This would work for all the cases except when a and b are adjacent. To account for this case, we need a second rule definition where we assume that a is before b.

```
swapLinkedAdj (a? b y z?) =
  a precedes b, b precedes y, y precedes z
  =>
  a precedes y, y precedes b, b precedes z
```

When applying the transformation on two elements in the GUI code, we still have to check whether the elements are adjacent or not, and which of the elements comes first in the list. We can then choose the correct transformation rule with the correct order of arguments. We discuss more on nullable placeholders in Chapter 6.

3.2.9 Inserting into and removing from a container

Another example of structural changes is that of inserting into and removing from a container of elements. To define transformation rules for such structural changes, we can specify the relations *isIn* and *isNotIn* for a container and an element with their accompanying predicates and code blocks that define how these relations are established.

```
relations
  (isIn) elem cont ::=
    test '''«cont».contains(«elem»)'''
    establish '''«cont».insert(«elem»)'''
    no unestablish
  (isNotIn) elem cont ::=
    test '''!«cont».contains(«elem»)'''
    establish '''«cont».remove(«elem»)'''
    no unestablish
```

We then specify for a containter and an element the rules *insert* and *remove*

```
rules
  insert (elem cont) = elem isNotIn cont => elem isIn cont
  remove (elem cont) = elem isIn cont => elem isNotIn cont
```

We can arguably improve the definitions of the rules by defining a procedure for unestablishing the relation *isIn*; we can use the establish procedure of `isNotIn`. This removes the need for isNotIn and allows us to give a simpler definition of the rules:

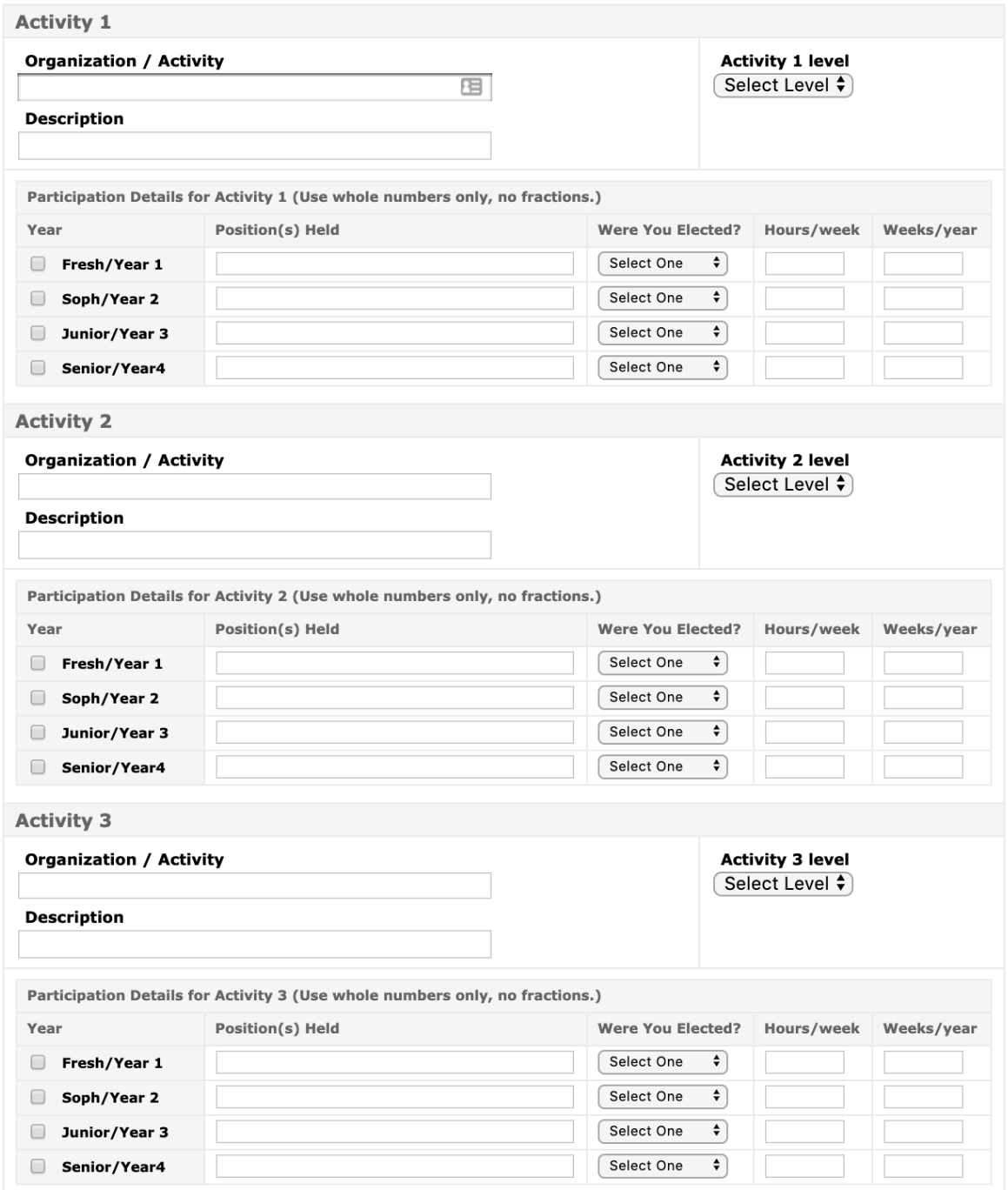
```
  insert (elem cont) = => elem isIn cont
  remove (elem cont) = elem isIn cont =>
```

3.2.10 ApplyTexas

While every example up to now has been fictional, our next example is a realistic GUI program that lacks support for structural changes. The GUI is one of several forms found on *ApplyTexas* [1], a website that provides admissions to institutions of higher education in Texas. In this particular form, the applicant fills in his or her extracurricular activities as part of an admission.

As seen in Figure 3.2, each activity in the form has 23 fields: text inputs, numerical inputs, and checkboxes. There can be up to ten activities, and the instructions ask the applicant to specify the activities in the order of importance. There are, however, no way of reordering the activities. Thus, the easiest way to move an activity up or down in the form, swapping it with another activity, is to manually copy and paste each field of the two activities. If the applicant wants to make several reorderings, the person is best off refreshing the website and starting over again. This hurts the users as they

Figure 3.2: Form for extracurricular acitivities in the ApplyTexas application.



either waste a lot of unnecessary time reprioritizing, or they avoid reprioritizing and submit the activities in a suboptimal order. We conjecture that the form lacks reordering capabilities because the developers of the application considered it to be too much work for the benefit.

Using our framework, we introduce reordering of activities by allowing each activity to be swapped with the activity above or below. For each activity, we create a corresponding JavaScript object with the fields `before` and `after`. The fields `before` and `after` of an activity a point, respectively, to the activities before and after a in the prioritized order.

In this case, the components are the activities, each consisting of both a model and view. We represent the component using a JavaScript object with two fields: `model` and `view`. The relation *isAbove* holds on two activities x and y if and only if x is directly above y. We define swapping rules for swapping two adjacent activity items, appending rules for appending an activity after the last activity, and removal rules for removing an activity from the list.

Using the DSL of our framework, we define the components and relations in the program in Figure 3.3, and in Figure 3.4 we define the transformation rules. Implementing reordering is now trivial: one can swap two adjacent activities a and b by applying one of the specified swapping transformations to the two components. If a and b are the only activities in the list, one can call the generated JavaScript function `swapAdjacentOnly` with the two components as the arguments. If there are more activities in the list, one of the other generated swapping functions can be called, such as `swapAdjacentAtBeginning` or `swapAdjacentInBetween`.

In the establish procedure of the relation `isAbove`, we position the view of the second placeholder `y` after the view of the first placeholder `x`. When we swap two activities, we move their entire HTML `div`-nodes without modifying the content of the nodes. Another way to perform the swap operation would be to swap the *values* of the 23 fields of the two activities involved, instead of swapping the *DOM nodes* corresponding to them. Implementing this would involve changing the establish procedure of the relation `isAbove`, and no changes to the swapping rules would be required. In both cases, the programmer can treat the activities and relations between them as a "list of elements", even though the operations of rearranging the elements have different semantics. This enables the programmer to view the dynamic structure in a uniform way, abstracting from the implementation details of relations between components.

Figure 3.3: Structural changes in the ApplyTexas GUI: specification of components, placeholders and relations.

```
components
  Activity <-> ''' 'before' in «Activity».model
    && 'after' in «Activity».model
    && «Activity».view instanceof HTMLDivElement '''
  Container <-> ''' «Container» instanceof HTMLDivElement '''

placeholders
  x, y, a, b: Activity
  cont: Container

relations
  (isAbove) x y ::=
    test '''«x».model.after === «y» && «y».model.before === «x»'''
    establish '''
      «x».model.after = «y»
      «y».model.before = «x»
      «x».view.parentElement = «y».view
      «x».view.parentElement
         .insertBefore(«y».view, «x».view.nextElementSibling) '''
    unestablish '''
      «x».model.after = null
      «y».model.before = null '''

  (isIn) x cont ::=
    test '''«cont».contains(«x».view)'''
    establish '''«cont».append(«x».view)'''
    unestablish '''«x».view.remove()'''
```

Figure 3.4: Structural changes in the ApplyTexas GUI: specification of transformation rules.

```
rules
  init (cont x) =
    => x isIn cont

  append (a x) =
    => a isAbove x

  swapAdjacentOnly (x y) =
    x isAbove y => y isAbove x

  swapAdjacentAtBeginning (x y b) =
    x isAbove y, y isAbove b
    => y isAbove x, x isAbove b

  swapAdjacentInBetween (a x y b) =
    a isAbove x, x isAbove y, y isAbove b
    => a isAbove y, y isAbove x, x isAbove b

  removeOnly (cont x) =
    x isIn cont =>

  removeAtBeginning (cont x a) =
    x isIn cont, x isAbove a =>

  removeAtEnd (cont a x) =
    x isIn cont, a isAbove x =>

  removeInBetween (cont a x b) =
    x isIn cont, a isAbove x, x isAbove b
    => a isAbove b
```

3.3 Summary

By defining rules as a transformation of relations, the rules are abstracted away from the mechanics of managing connections between components. Therefore, the rules clearly communicate the structural changes they are representing.

The beginning of this chapter discussed the problems of implementing structural changes in GUIs: components are connected to each other in ad-hoc ways, both in the GUIs model and view. A structural change must break and rebuild such connections; without guidance, this typically leads to unstructured code.

Our framework codifies the connections between components as relations and structural changes in a GUI as transformations of relations.

Chapter 4

Implementation

We built a working implementation of *WarmDrink*, our framework for specifying structural changes. In this chapter, we explain the implementation details as well as the development tools involved in the process.

4.1 Eclipse and Xtext

We developed a DSL, together with an accompanying *integrated development environment* (IDE), for writing WarmDrink programs. Programming languages have accompanying IDEs to improve the language users' development experience; IDEs provide syntax highlighting of code, code completion suggestions, error messages at the position of the errors in the code, analysis of code, and refactoring tools. Instead of implementing the entire IDE ourselves, we use a *language workbench* [12] to develop the IDE. A language workbench is a program for defining languages and IDEs for them, examples of language workbenches are Racket [14], JetBrains MPS [8], and Spoofax [20]. For our DSL, we use the language workbench *Eclipse Xtext* [13].

The first step to define a DSL in Xtext is to define a grammar using *Extended Backus-Naur form* [6] (EBNF). The grammar should be compliant to *ANTLR's recursive descent parsing algorithm* [24]; for instance, one of the limitations of these grammars is that left-recursive production rules, such as $Expr \rightarrow Expr + Expr$, are not allowed.

Consider the following example of a grammar rule in Xtext:

```
Relation: "(" name=ID ")" args+=[Placeholder]* "::="
    "test" testCode=JavaScriptCode
    ("establish" establishCode=JavaScriptCode)?
    ("unestablish" unestablishCode=JavaScriptCode)?
;
```

The name of the rule is `Relation`, as seen in the left-hand side above, and the right-hand side is the body of the rule. In Xtext grammar, we write asterisks (`*`) for expressions that occur zero to many times, question marks for optional expressions, and parentheses to group consecutive expressions. The elements `args`, `testCode`, `establishCode`, and `unestablishCode` are *features* of the rule. Features act as variables in rules, and they are populated with expressions for later access. The feature `testCode`, for instance, will be populated with a `JavaScriptCode` expression. Xtext allows specifying cross-references in grammar rules; `[Placeholder]` refers to an existing instance of `Placeholder`.

From a language grammar Xtext generates an *Eclipse Modelling Framework [16] Ecore model*. This model is an object model and is populated during parsing of the code. Using a general-purpose JVM language, such as Java or Xtend [7], the language developer can process the model further and, for instance, define code *validations* that report on issues in the code and define *quickfixes* that edit the code whenever they are triggered. One can also define code generators that perform model-to-model or model-to-text transformations [22]. In our case, we want to transpile our DSL to JavaScript code and, therefore, we use a model-to-text transformation. To transform the model, we use Xtend's *template strings*: inside a string, we define expressions that evaluate to strings; these expressions are wrapped in guillemet quotes («»). On the evaluation of a template string, its inner expressions are evaluated to strings and inserted into the final string at their initial positions.

Based on the grammar for our DSL and the Ecore model-to-JavaScript transformation we defined, Xtext builds an Eclipse-based IDE.

4.2 Code generation

Our implementation of WarmDrink targets JavaScript because of the popularity of the language in GUI programming [2]. However, the concepts of WarmDrink are not specific to any language, and we can target other imperative languages by implementing code generators for those languages.

In this section, we will explain how each part of a WarmDrink DSL program is transpiled to JavaScript. For each class in the model, we define a `compile`-method that returns the generated code for an instance of that class. If an instance contains child nodes, the `compile`-method of the instance generates the code for the child nodes by calling their `compile`-methods. The code for the entire program is, thus, generated by calling `compile` on the root of the model. We implemented the code generator using Xtend, an imperative programming language similar to Java.

4.2.1 Specification of procedures

JavaScript procedures in the WarmDrink DSL are instances of the class `RichString` [11], and each `RichString` contains a list of `expression`-nodes. Each `expression` node has the subtype `RichStringParts` if it is a reference to a placeholder, or the subtype `RichStringLiteral` if it is a plain JavaScript code fragment. WarmDrink transpiles `RichStringLiteral`s by removing any triple quotes and guillemets, and `RichStringPart`s to the name of their placeholders.

```
def CharSequence compile(RichString richString) {
  ''' «FOR expression : richString.expressions»
        «switch expression {
          RichStringLiteral: expression.compile
          RichStringPart: expression.compile
        }»
      «ENDFOR»'''
}
def CharSequence compile(RichStringLiteral richStringLiteral) {
  richStringLiteral.value.replaceAll("«", "")
    .replaceAll("»", "").replaceAll("'''", "")
}
def CharSequence compile(RichStringPart richStringPart) {
  richStringPart.ref.name
}
```

Thus, the JavaScript procedure

```
'''referToSameValue(«b».model.vs.prev, «a».model.vs.end)'''
```

is parsed to the object model

```
RichString
  [ RichStringLiteral("referToSameValue(")
    RichStringPart (Ref "b")
  , RichStringLiteral ".model.vs.prev.value, "
  , RichStringPart (Ref "a")
  , RichStringLiteral ".model.vs.end)"]
```

which, in turn, is transpiled to

```
referToSameValue(b.model.vs.prev, a.model.vs.end)
```

Note that the JavaScript code in the DSL and the transpiled code are almost the same, but having the placeholders in guillemets in the DSL allows the programmer to use IDE tools on the placeholders. For instance, our IDE can rename placeholders and report on invalid references to placeholders in JavaScript procedures.

4.2.2 Specification of JavaScript code blocks

The first part of a WarmDrink DSL program is an optional JavaScript code block containing imports and helper functions. The purpose of this code block is to keep the other procedures concise, as they can take advantage of predefined classes and functions. To define a JavaScript code block, the programmer writes the keyword `javascript`, followed by JavaScript code wrapped in triple quotes (`'''`). The JavaScript code is parsed as an instance of `RichString`.

The code block is (trivially) transpiled to JavaScript by calling the `compile`-method on the `RichString`-instance as described above. We give an example of a code block and the resulting generated JavaScript code:

```
javascript
  ''' require('someModule') '''
```

transpiles to

```
require('someModule')
```

Figure 4.1: Transpiling components to JavaScript.

```
def CharSequence compile(Component component) {
 '''
  function $WD_expect_«component.name»(«component.name») {
    if ( !(«component.jsCode.code.compile») )
      throw Error(
        "Expected component '«component.name»', " +
        "but component of other type passed");
  }
  '''
}
```

4.2.3 Specification of components

In the component section of a WarmDrink DSL program, the programmer declares the component types that are used in the program. Each component type t has an accompanying *type-checking* procedure, which is a boolean function that returns true if and only if its parameter is a component of type t. The procedure is transpiled to a JavaScript function `$WD_expect_`t that takes a component as its parameter and throws an error if the component has the wrong type. Whenever WarmDrink rule transformations are applied to JavaScript objects, the objects are type-checked by passing them as arguments to the type-checking functions.

The need for such type-checking functions comes from the fact that JavaScript is a dynamically typed language. If WarmDrink targeted a statically typed language, such as Java or TypeScript [9], the application programmer would specify the component types directly, making the type-checking procedures redundant.

Note that many of the functions generated by WarmDrink, including the function `$WD_expect_`t, are prefixed with `$WD_`. These functions are not intended to be called by application programmers, and they are only called from within other WarmDrink-generated functions. The prefix serves as a warning to the programmer and prevents name collisions between WarmDrink-generated code and the user code.

The function for transpiling WarmDrink component types to type-checking functions is defined in Figure 4.1. The parameter of the type-checking function uses the same name as the type that is being defined. Thus, in the JavaScript code, the programmer refers to the parameter by using that name. For example, the following component type declaration

```
  Event <->
    '''
      «Event».model.constructor.name === "Component"
      && «Event».view instanceof HTMLDivElement
    '''
```

generates the following type-checking function:

```
function $WD_expect_Event(Event) {
  if ( !(
        Event.model.constructor.name === "Component"
        && Event.view instanceof HTMLDivElement
   ))
    throw Error(
      "Expected component 'Event', " +
      "but component of other type passed");
}
```

In the generated code, the type-checking procedure for `Event` is used as a condition in the if-statement.

4.2.4 Specification of placeholders

In the placeholders section of a WarmDrink DSL program, the application programmer defines component placeholders that are used as parameters in the relation specifications and transformation rules. Below is an example of the placeholders section:

```
placeholders
  a: Event
  b: Event
```

This code snippet declares the placeholders `a` and `b`, both standing for components of type `Event`.

We discuss below how WarmDrink generates JavaScript functions for relation specifications and transformation rules. Those functions expect their parameters to be components that the types of their placeholders specify. When WarmDrink generates those functions, the placeholders' types determine which type-checking function to call on each parameter. No JavaScript code is generated for the placeholders themselves.

4.2.5 Specification of relations

In the relation section of a WarmDrink DSL program, the application programmer defines the relation specifications. A relation specification contains a list of component placeholders, a `test` procedure testing whether the relation holds, an optional `establish` procedure that establishes the relation and an optional `unestablish` procedure that unestablishes the relation. The example below showcases a specification of a relation `myRel` defined on two placeholders `a` and `b`.

```
relations
  (myRel) a b ::=
    test ''' JavaScript test code here (1) '''
    establish ''' JavaScript establish code here (2) '''
    unestablish ''' JavaScript unestablish code here (3) '''
```

The following Xtext grammar rule is used to parse relation specifications:

```
Relation:
  "(" name=ID ")" args+=[Placeholder]* "::="
  "test" testCode=JavaScriptCode
  ("establish" establishCode=JavaScriptCode)?
  ("unestablish" unestablishCode=JavaScriptCode)?
;
```

The feature `args` here is a list of placeholders, and the features `testCode`, `establishCode` and `unestablishCode` hold the code procedures associated with a relation specification.

For each relation specification r, WarmDrink generates three JavaScript functions `$WD_test_`r, `$WD_establish_`r and `$WD_unestablish_`r. Below we give listings of the functions that will be generated for the sample relation `myRel`, and explain the code generation process in Xtend.

```
function $WD_test_myRel(a, b) {

  // check arguments
  $WD_expect_Event(a);
  $WD_expect_Event(b);

  // test relation predicate
  return JavaScript test code here (1);
}
```

The function `$WD_test_myRel` generated from the relation `myRel` has two parameters, `a` and `b`. The function checks that the parameters have the same type as their corresponding placeholders, in both cases the expected type is `Event`. The function performs the type-checks by calling `$WD_expect_Event` on both `a` and `b`, so that an error is thrown if the type-check fails. Then, the function checks whether `myRel` holds by running the `test` JavaScript code provided in the relation specification. The function returns `true` if the relation holds, and `false` otherwise.

Below is the code to generate the `test` function for a relation specification:

```
1  def CharSequence compile(Relation relation) {
2    '''
3    function $WD_test_«relation.name»(
4      «FOR arg : relation.args SEPARATOR ','»«arg.name»«ENDFOR») {
5
6      // check arguments
7      «FOR arg : relation.args»
8        $WD_expect_«arg.component.name»(«arg.name»);
9      «ENDFOR»
10
11     // test relation predicate
12     return «relation.testCode.code.compile»;
13   }
14   ...
```

The name of the relation is used in the name of the generated test function, as seen at line 3. At line 4, the parameters of the generated function are generated using the names of the placeholders. Those parameters are type-checked by the code generated at lines 7–9. Line 12 outputs the statement to check whether the relation holds using the given `test` code.

Figures 4.2 and 4.3 show the generated `establish` and `unestablish` functions, respectively, for the relation `myRel`. The bodies of the two generated functions are quite similar and, therefore, we only explain the `establish` function and how it is generated. The function `$WD_establish_myRel` establishes the relation `myRel` on the two parameters `a` and `b`. First, it type-checks `a` and `b` by passing them as arguments to their type-checking functions. It then establishes the relation on the two parameters by running the `establish` code given in the specification of `myRel`. Finally, the generated function checks that `myRel` has indeed been established by calling the testing function `$WD_test_myRel` generated for the relation. An error is thrown if the relation does not hold.

Figure 4.2: The generated JavaScript function for establishing the relation `myRel`.

```
function $WD_establish_myRel(a, b) {

  // check arguments
  $WD_expect_Event(a);
  $WD_expect_Event(b);

  // establish relation
  JavaScript establish code here (2)

  // test relation predicate
  if (! $WD_test_myRel(a, b) ) {
    throw Error("Relation predicate not satisfied "
    + "after the relation has been established");
  }
}
```

Figure 4.3: The generated JavaScript function for unestablishing the relation `myRel`.

```
function $WD_unestablish_myRel(a, b) {

  // check arguments
  $WD_expect_Event(a);
  $WD_expect_Event(b);

  // unestablish relation
  JavaScript unestablish code here (3)

  // test relation predicate
  if ( $WD_test_myRel(a, b) ) {
    throw Error("Relation predicate still satisfied "
    + "after the relation was unestablished");
  }
}
```

Below is the Xtend code that generates the `establish` function for a relation specification:

```
1  '''
2  ...
3  function $WD_establish_«relation.name»(
4    «FOR arg : relation.args SEPARATOR ','»«arg.name»«ENDFOR») {
5
6    // check arguments
7    «FOR arg : relation.args»
8      $WD_expect_«arg.component.name»(«arg.name»);
9    «ENDFOR»
10
11   // establish relation
12   «relation.establishCode.code.compile»
13
14   // test relation predicate
15   if (! $WD_test_«relation.name»
16     («FOR arg : relation.args SEPARATOR ','»«arg.name»«ENDFOR»)
17   ) {
18     throw Error("Relation predicate not satisfied "
19     + "after the relation has been established");
20   }
21 }
22 ...
```

The function signature is generated at lines 3–4 using the relation name and the placeholder names. Lines 6–9 output calls to the type-checking functions for the placeholders. Lines 11–12 generate the code for establishing the relation using the `establish` part of the relation specification. The code for testing whether the relation holds is generated at lines 14–20.

4.2.6 Specifications of transformation rules

In the rules section of a WarmDrink DSL program, the programmer defines the rule transformations for the components. A transformation rule is comprised of a list of placeholders, a list of pre-connections, and a list of post-connections. Pre-connections are the relations that should hold before the transformation, and post-connections are the relations that should hold after the transformation. The example below showcases a definition of a transformation rule `tr` defined on three placeholders `a`, `b`, and `c` of the type `Event`, assuming the relation `myRel` from above was defined:

```
rules
  myRule (a b c) = a myRel c => a myRel b, b myRel c
```

Transformation rules are parsed according to the following Xtext grammar rule:

```
Rule: name=ID "("args+=[Placeholder]*")" "="
  (preConnections += Connection ("," preConnections +=
     ↪ Connection)*)?
  "=>"
  (postConnections += Connection ("," postConnections +=
     ↪ Connection)*)?
;
```

The feature `args` contains the placeholders of the rule, and the features `preConnections` and `postConnections` contain the pre-connections and post-connections of the rule, respectively. Both `preConnections` and `postConnections` can be empty.

The generated JavaScript function from the sample rule `myRule` is given in Figure 4.4. It takes three parameters `a`, `b`, and `c`, and type-checks them. It checks that the relation `myRel` holds for `a` and `c`, and aborts the operation otherwise. Then, it unestablishes the relation `myRel` on `a` and `c`, and establishes `myRel` on `a` and `b` and on `b` and `c`. Finally, the function checks that the established relations hold, and prints a warning otherwise.

Figure 4.5 contains the code for transpiling transformation rules into JavaScript functions. Unlike JavaScript functions generated from the previous WarmDrink DSL sections, JavaScript functions generated from transformation rules are intended to be invoked by the application programmer. Therefore, the generated function has no `$WD_`-prefix, and the name of the generated function is the same as the name of the transformation rule.

As before, lines 2–3 generate the function signature, and lines 4–7 output code for type-checking the parameters. Lines 9–19 generate code for testing the pre-connections that are later unestablished by the code generated by lines 21–26. Establishing the post-connections of the transformation rule is handled by the code in lines 28–33. Finally, lines 35–43 generate JavaScript code that checks whether the post-connections have indeed been established.

Note that at lines 12, 24, 31 and 38, we output JavaScript code that tests, establishes, and unestablishes relations. We explain now how the method `compile` is defined for the variable `connection` in each of these lines. This variable has the type `BinaryConnection`,

Figure 4.4: The JavaScript function generated from the rule `myRule`.

```
function myRule(a, b, c) {
  // typecheck arguments
  $WD_expect_Event(a);
  $WD_expect_Event(b);
  $WD_expect_Event(c);

  // test preconditions
  preConditions = true
    && $WD_test_myRel(a, c)

  if (!preConditions) {
    console.log("One of the preconditions doesn't hold in myRule");
    return;
  }

  // unestablishing old relations
  $WD_unestablish_myRel(a, c)

  // establish new relations
  $WD_establish_myRel(a, b)
  $WD_establish_myRel(b, c)

  // test postconditions
  postConditions = true
    && $WD_test_myRel(a, b)
    && $WD_test_myRel(b, c)

  if (!postConditions) {
    console.log("Postcondition doesn't hold in rule myRule");
  }
}
```

Figure 4.5: Transpiling rules to JavaScript.

```
def CharSequence compile(Rule rule) {'''
  function «rule.name»(
    «FOR arg : rule.args SEPARATOR ','»«arg.name»«ENDFOR») {
    // typecheck arguments
    «FOR arg : rule.args»
      $WD_expect_«arg.component.name»(«arg.name»);
    «ENDFOR»

    // test preconditions
    preConditions = true
    «FOR connection : rule.preConnections»
      && «connection.compile("test")»
    «ENDFOR»;

    if (!preConditions) {
      console.log(
        "One of the preconditions doesn't hold in rule
          ↪ '«rule.name»'");
      return;
    }

    // unestablishing old relations
    «FOR connection : rule.preConnections»
      «IF connection.getRel().unestablishCode !== null»
        «connection.compile("unestablish")»;
      «ENDIF»
    «ENDFOR»

    // establish new relations
    «FOR connection : rule.postConnections»
      «IF connection.getRel().establishCode !== null»
        «connection.compile("establish")»;
      «ENDIF»
    «ENDFOR»

    // test postconditions
    postConditions = true
    «FOR connection : rule.postConnections»
      && «connection.compile("test")»
    «ENDFOR»;

    if (!postConditions) {
      console.log("Postcondition doesn't hold in rule
        ↪ '«rule.name»'");
    }
  } '''
}
```

which represents a relation on two component placeholders. The following Xtext grammar rule is used to parse such binary relations; it has the feature `rel` that represents the relation, as well as the features `arg1` and `arg2` that represent the placeholders.

```
BinaryConnection:
    arg1=[Placeholder] rel=[Relation] arg2=[Placeholder]
;
```

The Xtend function `compile` given below takes a `BinaryConnection` and a string `modifier` as arguments, where `modifier` is either `"test"`, `"establish"` or `"unestablish"`. The function generates a JavaScript function call by using the modifier, name of the relation, and name of the placeholders. For instance, to generate the function call to establish the connection `"a precedes b"`, we call the `compile` function on the connection with the string `"establish"` as argument: `connection.compile("establish")`. The function returns the string `"$WD_establish_precedes(a, b)"`.

```
def dispatch CharSequence compile(BinaryConnection binaryConnection,
  String modifier) {
  '''$WD_«modifier»_«binaryConnection.rel.name»(
      «binaryConnection.arg1.name»,
      «binaryConnection.arg2.name»
    )'''
}
```

We have thus explained the implementation details of the WarmDrink DSL parser, Eclipse-based IDE, and code generator to JavaScript.

Chapter 5

Structural changes in constraint system-powered GUIs

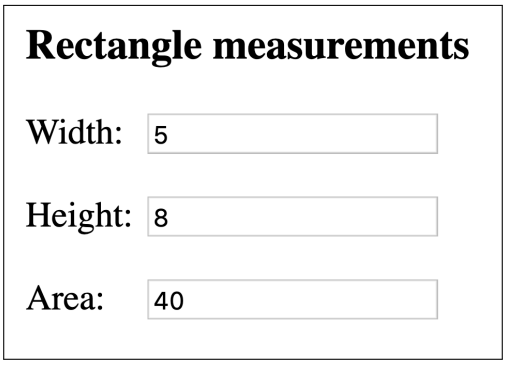
In this chapter, we will explain how WarmDrink is harnessed to manage structural changes in GUI applications based on constraint systems. We first explain what a constraint system is and introduce *HotDrink*, a software library for defining and solving constraint systems in GUIs. We then describe an example application that showcases the use of *HotDrink* and *WarmDrink* together.

5.1 Multi-way dataflow constraint systems

The *dataflow* of a program is a graph of variables, where directed edges between variables express flows of data, or dependencies between variables. More specifically, if a variable a is involved in the computation of a variable b, and b must be updated whenever a is updated, then b is dependent on a; we say that data *flows* from a to b.

GUI programs with interactive forms often exhibit dataflow. An example of such a GUI is the interactive tax calculator on *Skatteetaten*s website [4], where users fill information about their personal economics in a form to calculate their total taxes. Whenever fields in the form are updated, a table of intermediate results, such as common tax and national insurance contributions, is updated. The total tax, which is the sum of all the intermediate results, is also computed. In the calculator, data flows from the input fields in the form to the table of intermediate results, and subsequently from the table to the tax result.

Figure 5.1: Example GUI form for rectangle measurements.



In dataflow programming, the programmer can specify such functional dependencies between variables, also called one-way dataflow constraints [25]. Sometimes programs exhibit dataflows to multiple directions: at one point of a program execution a depends on b but at another point b depends on a. If variables are dependent on each other so that data flows more than one way, the flow is a *multi-way dataflow*.

We give in Figure 5.1 an example GUI that exhibits multi-way dataflow: an interactive form for detecting the measurements of a rectangle. The GUI's model has three variables: *width*, *height* and *area*, and for each variable, the view, a simple form, has a corresponding numerical input field. There is a binding between the input fields and the variables; users can edit the value of a variable by editing the number in the corresponding input field. When users edit one of the variables, the other two variables are automatically updated so that the equation

$$area = width * height$$

is satisfied. The mechanics of the GUI can update any of the three variables to satisfy the equation.

Assuming that *width*, *height* and *area* are 3, 5 and 15, respectively, imagine the following user scenario: a user edits *area* to be 30, and thus the mechanics of the GUI satisfies the equation by updating *width* to be 6 and leaving *height* at 5. Then, the user updates the *width* to be 10. One way the mechanics can satisfy the equation at this point is by updating *area*. Since changes to *width* can affect *area* and changes to *area* can affect *width*, data flows in both directions between *width* and *area*. The dataflow of this GUI is, therefore, a multi-way dataflow.

A dependency cycle in the dataflow occurs if a value a (transitively) depends on another value b, and b (transitively) depends on a. If there are no such cycles, the dataflow graph has the structure of a directed acyclic graph (DAG).

A common approach for defining the reaction to user input in graphical user interfaces is using *event listeners*. Event listeners are methods attached to widgets or components that users interact with, such as text inputs and buttons. On user interaction, these methods are triggered with the interaction event as the argument, and they define the desired behavior for that interaction; it can be, for instance, updating the internal data model, making an update to other widgets in the user interface, or sending a request to a server.

We can use event listeners to propagate data in one-way dataflows. If a user interacts with some widget w, the event listener of w can take care of updating the widgets ws that depend on w. This will trigger the event listeners of ws to in turn update the widgets that depend on ws. These updates continue transitively until there are no more dependencies.

Propagating data in a multi-way dataflow, however, is complex using event listeners. Event listeners cannot push data to their dependents, as data may end up propagating in a cycle. Each listener, therefore, needs to update all the transitive dependents whenever it is triggered. This breaks the *separation of concerns* [23, p. 5], a design principle of programming that states that each section of a program should only have one concern. Programs that adhere to this principle have the advantage of being *modular*; changes can easily be made to one section, or *module*, of the program without the need for other modules to change. If a GUI has an event listener bound to a variable v in a module m, and the listener method updates all the transitive dependents on v, including a variable w in another module n, the program is not modular: a change can be made to the module n that affects w. Consequently, the event listener method for v in the module m needs to change, and thus the program does not adhere to the separation of concerns principle.

Furthermore, it is nontrivial in the listener methods to determine which way data should flow if there are multiple alternatives. The program could remember the most recently edited variables so that it can prioritize flows that update less recently edited variable, if such flows exist. For instance, in the rectangle example above, a user may first update *height*, then *area*, and then *width*; the intuitive behavior would likely be that the mechanics update *height*, and not *area*, after the last update. Finally, if concurrency is in play — event listeners can run asynchronously — ensuring that unpredictable and erroneous dataflows cannot occur becomes difficult.

Since using event listeners to propagate data in GUIs that exhibit multi-way dataflows is complex, it is better to have *multi-way constraint systems* propagate data in these GUIs. A *constraint* is a relation defined on a set of variables. When the relation holds on the variables, the constraint is enforced. More specifically, for a constraint r on a set of variables $v_1, ..., v_n$, r is enforced iff the values $a_1, ..., a_n$ of the variables $v_1, ..., v_n$, respectively, are such that $\langle a_1, ..., a_n \rangle \in r$. An example of a constraint is found in our example GUI above: the constraint on the variables *area*, *width* and *height* is defined using the following relation *rect*:

$$\langle area, width, height \rangle \in rect \iff area = width * height$$

Thus, whenever *area* is the product of *width* and *height*, the constraint is enforced.

A *constraint system* is a set of constraints, and the constraint system is *solved* whenever all the constraints in the system are enforced. Constraint systems in GUIs are solved repeatedly; each time a variable in a constraint system is updated, leading to one or more constraints being violated, the constraint system has to be solved by updating the other variables in the system.

5.2 HotDrink

HotDrink [15] is a JavaScript library for GUI programming with a hierarchical multi-way dataflow constraint system as a core. To utilize HotDrink in GUI programs, a programmer defines the variables and constraints of the GUI and adds them to a constraint system managed by the library. The variables and constraints are defined either in JavaScript or in a custom DSL for HotDrink that transpiles to JavaScript. When a variable in the GUI is updated, possibly violating a constraint, the library automatically solves the constraint system [19] by finding a dataflow that updates all the variables that need to be updated to enforce all the constraints. The library then computes new values to those variables according to the dataflow. If the library finds more than one such dataflow, it chooses the dataflow that leaves the most recently edited variables unchanged; the behavior of keeping the most recent edits is often intuitive and preferred by the user. If no such dataflow can be found, then there are no ways to solve the constraint system, and thus none of the variables are updated.

A *variable* in HotDrink contains a value, and the application programmer can update a variable `v` with a new value `newValue` by calling the method `v.set(newValue)`. To get notified of updates to v, the programmer can *subscribe* to `v` by calling the method `v.subscribe(cb)`, where `cb` is a *callback function*; the `cb`-function is called with a new value as its argument whenever the variable is updated. The programmer can establish a *binding* between a variable and a GUI component so that either the variable is updated whenever the component is updated, or the component is updated whenever the variable is updated. For a component to update whenever a variable is updated, the programmer subscribes to the variable and updates the component in the callback function of the subscription. For a variable to update whenever the user interacts with a component, the programmer attaches an event listener to the component and updates the variable in the event listener method. If there is a binding between a variable and a component such that whenever one of them updates the other one is updated, we say there is a *two-way* binding between the variable and the component.

Variable declarations in the HotDrink DSL are similar to variable declarations in JavaScript: a variable is declared using the keyword `var` followed by the name of the variable. A value is, then, optionally assigned to the variable using an equals sign and the value, and the statement ends with a semicolon. More variables can be declared in one statement by comma-separating the variable declarations. In the following code example, we declare the variable x and y using the HotDrink DSL:

```
var x=5, y;
```

The variable x has initially the value 5, and y is initially undefined.

A constraint on a set of variables is specified using *constraint satisfaction methods* in HotDrink. Assuming a constraint c on a set of variables vs, a constraint satisfaction method of c is a method that enforces c when it is executed. In the specification of the method, the programmer specifies two subsets *input* and *output* of vs; the method will take *input* as its argument set and the values that the method returns are used to update *output*. We say that the method *reads* from *input* and *writes* to *output*.

As an example on declaring constraints in HotDrink, consider the constraint *rect* from the example above defined on the variables *width*, *height* and *area*: *rect* is enforced whenever *area* is the product of *width* and *height*. By knowing the value of two of the three variables, we can compute a new value for the third variable such that *rect* becomes

enforced. For instance, if we know *area* and *height*, we can update *width* to be *area* divided by *height*. Thus, we have three ways, or methods, to enforce *rect*. We declare the constraint *rect* with the three methods using the HotDrink DSL:

```
var width=3, height=5, area=15;
constraint rect {
  m1 (width, height -> area) => width * height;
  m2 (area, width -> height) => area / width;
  m3 (area, height -> width) => area / height;
}
```

In the DSL code, the constraint `rect` has three constraint satisfaction methods `m1`, `m2` and `m3`. The code in the parenthesis, after the method names, defines the *input* and *output* variables of the methods; `m1`, for instance, reads from *width* and *height* and writes to *area*. The code after the double arrow is the body of the method. The return value of the method is written to the output variables of the method. If there are more than one output variable, the method must return an array. HotDrink can enforce the constraint *rect* by executing either `m1`, `m2` or `m3`. Note that HotDrink can enforce *rect* without knowing the conceptual relation of *rect*; it only knows how to enforce the relation.

What methods the programmer uses in the constraint declarations depends on how the programmer wants data to flow; the programmer could, in the example above, declare the constraint `rect` to only contain the method `m1`, that is, only telling how to compute the *area* when *width* or *height* change. In that case, regardless of whether the user edits *width* or *height*, area will always be updated.

A *component* in HotDrink is a composition of variables and constraints, and a component typically corresponds to a group of GUI widgets. Each variable and constraint in HotDrink is defined in a component, and they are owned by it too. A programmer can create a copy of a component by *cloning* it. Cloning a component is useful when the component corresponds to a GUI widget that has multiple occurrences, such as list items.

In addition to variables and constraints, components can have *variable references*. Variable references are nullable references to variables. They can be used in constraint methods as if they were variables, but whenever a variable reference is null, the constraints that use the reference are inactive and ignored by the constraint solver. Whenever all variable references in a constrain refer to a variable, the constraint is active. The programmer can *connect* two components by referring a variable reference in one component

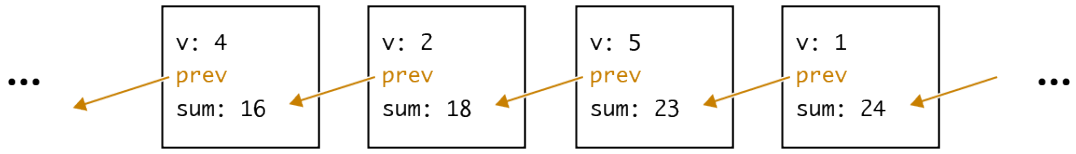


Figure 5.2: Connected HotDrink components.

to a variable in the other. Syntactically, variable references are declared as variables, except that the name of a variable reference is prefixed with an ampersand sign. Variable references cannot be initialized with values.

We provide an example to show the use of variable references: assume a sequence of HotDrink components $c_1, ..., c_n$, where each component has a variable v with a number, and we want to accummulate the sum of all the numbers in the sequence. In other words, each component c_k, where $1 \leq k \leq n$, should hold the the sum of the v variables of $c_1, ..., c_k$. We do this by adding a variable reference *acc*, a variable *sum* and a constraint *accumulate* in each of the components:

```
var v=0, &acc, sum;
constraint accumulate {
  m (acc, v -> sum) => acc + v;
}
```

In the component specification above, we initialise the value of v to 0, and we initialise *acc* to null and *sum* is left undefined. Whenever the variable reference *acc* refers to a variable, the constraint *accumulate* is activated and the variable *sum* is updated to be $acc + v$. We connect each component c_k, where $1 < k \leq n$, to the preceding component c_{k-1} by referring the variable reference acc_k to the variable sum_{k-1}, as illustrated in Figure 5.2. In order to compute *sum* of the first component in the sequence, we refer *prev* of c_1 to a fixed variable with the value 0. Whenever v in any component is updated, the *sum* of that component and the rest of the components in the sequence are updated as well.

When a HotDrink component has a variable reference that refers to a variable in another component, we say that there is a *connection* between these components. Structural changes in GUIs with connected HotDrink components involve disconnecting and connecting components. This is inconvenient for the programmer: structural changes are high-level operations, but the programmer must manage low-level connections to perform these operations.

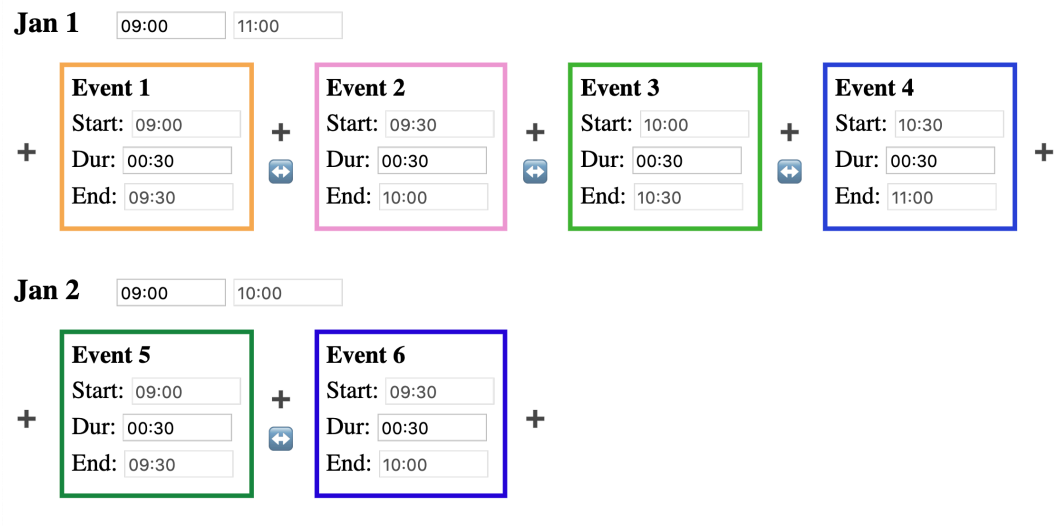


Figure 5.3: The GUI of the event scheduler example.

5.3 Example

We have made a constraint system-powered GUI that supports structural changes. The components in the GUI have connections to other components, and the structural changes in the GUI affect these connections: they tear down old and set up new connections between components. We use HotDrink to specify the constraints of the GUI and to enforce the constraints, and to specify how components are connected. We use WarmDrink to specify the structural changes in the GUI.

5.3.1 The scheduler

The GUI program is an event scheduler: the GUI is illustrated in Figure 5.3. The scheduler has a sequence of days, and each day has a sequence of events. The user can add new events to the schedule, remove events, and reorder the events. Each event has a duration which can be edited by the user, and the event has a start time and end time that are updated when the user edits the durations of the events. Each day in the scheduler has a start time and end time; the start time can be edited by the user, and the end time shows the end time of the last event of the day. If the last event of a day ends after midnight, it is automatically moved to the following day.

The GUI is implemented using the two libraries HotDrink and WarmDrink. The variables of the GUI and the constraints between the variables are specified with and

managed by HotDrink; HotDrink solves the constraint system of the GUI whenever a variable is updated by the user, and the library automatically updates the variables that need to change in order to solve the constraint system. The structural changes that the GUI supports, such as inserting events and swapping events, are implemented using WarmDrink. We specify the relations between the components and the transformation rules using the WarmDrink DSL. WarmDrink generates JavaScript functions that we call with component arguments to perform transformations.

5.3.2 Setup constraint system using HotDrink

There are two main types of components in the scheduler: the components *day* and *event*. The variables, variable references, and constraints of the components are specified using the HotDrink DSL.

Here is the specification of a day component in the HotDrink DSL, excluding irrelevant parts of the specification:

```
const day = hd.component`
  var start=new Time("00:00"), end, &lastEvent, ...;
  constraint lastToEnd {
    m1(lastEvent -> end) => lastEvent;
  }
  ...
`;
```

The code specifies the component *day* to have the variables *start* and *end*, and the variable reference *lastEvent*. We want the value of *end* to be equal to the end time of the last event, and we do this by using the variable reference *lastEvent* with the constraint *lastToEnd*: we intend *lastEvent* to refer to the ending time of the last event of the day. If the sequence of events is empty, *lastEvent* is `null`. We specify the constraint *lastToEnd*, which is a constraint on *lastEvent* and *end*, to be enforced by setting *end* equal to *lastEvent*. Thus, the value of the end time of the last event is forwarded to the variable *end*.

Figure 5.4 show the specification of an event component in the HotDrink DSL. The component *event* has the variables *start*, *duration* and *end*: we want the value of *start* to be the end time of the event that precedes this event or, if this event is the first event of the day, the start time of the day. Similarly as in the day component, we forward

Figure 5.4: Specification of an event component in the HotDrink DSL.

```
const event = hd.component`
  var &prev, start, duration=new Time("00:30"), end, title=10;
  constraint prevToStart {
    m1(prev -> start) => prev;
  }
  constraint sde {
    m1(start, duration -> end) => start.add(duration);
  }
`;
```

the start time to the variable *start* by using a variable reference *prev* and the constraint *prevToStart*, and we refer *prev* to the correct variable. The variable *duration* is the duration of the event and can be edited by the user. The variable *end* is the ending time of the event, it is computed from the variables *start* and *durations*; *end* is the time in *start* added to the time in *duration*. This dependency is handled by the constraint *sde*.

From the specifications above, we see that there are connections between the components in our GUI: the first event of each day is connected to the day through the variable reference *prev* of the event component, and rest of the events are connected to the previous events that day through the same variable reference *prev*. Furthermore, days are connected to the last event in that day through the variable reference *lastEvent* of the day component. Whenever we add, remove or reorder events we have to update these connections.

5.3.3 Implementing structural changes with WarmDrink

While there is a one-to-one mapping between the HotDrink components and the WarmDrink components in our GUI, the HotDrink component and WarmDrink component are not the same: a WarmDrink component in our GUI contains both a model and a view; the WarmDrink component is, conceptually, a pair consisting of a HotDrink component and a group of HTML DOM nodes. Concretely, a WarmDrink component is a JavaScript object with three properties: the property `type` is a string containing the type name, such as `"Event"` or `"Day"`. The property `model` is the HotDrink component, and the property `view` is the HTML DOM node of the component.

Here, we define the component `Event` in the WarmDrink DSL:

```
Event <->
  '''
    «Event».type === "Event"
    && «Event».model.constructor.name === "Component"
    && «Event».view instanceof HTMLDivElement
  '''
```

The component specification above has a procedure which (dynamically) type-checks event components: a JavaScript value is of type `Event` if it is a JavaScript object with the three properties `type`, `model` and `view`, where all three have the correct types, and the propery `type` of the object has the value `"Event"`.

The other component specifications in this example are similar to the specification of `Event`; the only difference is the content of the property `type`. Here is the specification of the component `Day`.

```
Day <->
'''
  «Day».type === "Day"
  && «Day».model.constructor.name === "Component"
  && «Day».view instanceof HTMLDivElement
'''
```

Using these component specifications, we define some placeholders that we can use in the relation specifications and transformation rules:

```
placeholders
  a, b, c, d: Event
  day: Day
```

The relation between a day and its first event is defined here in the WarmDrink DSL:

```
(firstOf) a day ::=
  test
    '''referToSameValue(«a».model.vs.prev, «day».model.vs.start)'''
  establish
    '''
      «a».model.vs.prev = «day».model.vs.start;
      cs.update();
      const eventList = «day».view.querySelector("[data-event]");
      eventList.insertBefore(«a».view, eventList.firstElementChild);
    '''
```

Figure 5.5: The relation specification `precedes`.

```
(precedes) a b ::=
  test
    '''referToSameValue(«b».model.vs.prev, «a».model.vs.end)'''
  establish
    '''
      «b».model.vs.prev = «a».model.vs.end;
      cs.update();
      «a».view.parentElement
         .insertBefore(«b».view, «a».view.nextElementSibling);
    '''
```

The name of the relation is `firstOf`, and the relation is defined for the placeholders `a` and `day`, which have the component types `Event` and `Day`, respectively. The test procedure of the relation tests whether the relation holds by checking if the value of the variable reference *prev* of `a` is the same as the value of the variable *start* of `day`. Because the property `value` of a HotDrink variable is a JavaScript object, we are, in fact, checking that the memory addresses of the two objects are the same. To establish the relation `firstOf`, we first point the variable reference *prev* of `a` to the variable *start* of `day` and notify the constraint system `cs` of the change. Then we acquire the DOM node that contains the event views of `day`, and insert the view of `a` as the first child of the DOM node. If the view of `a` already existed somewhere else in the DOM, it would automatically be removed from its previous position as DOM nodes only exists at one position at a time. We do not specify an unestablish procedure in this relation specification.

Another relation specification in our application is `precedes` defined in Figure 5.5. The relation `precedes` is defined for two placeholders `a` and `b` that both have the component type `Event`. The test and establish procedures are defined similarly as in the previous relation: the test procedure checks whether the relation holds by checking if the variable reference *prev* of `b` refers to the same value as the variable *end* of `a`. The establish procedure points *prev* of `b` to *end* of `a`, and inserts the view of `b` so that it follows the view of `a`.

In addition to the relation specifications `firstOf` and `precedes`, we specify other relations, such as `eventEndOf`. We use similar code as with `firstOf` and `precedes` to specify the rest of the relations between our components.

Using the placeholders and relation specifications defined above, we specify the transformation rules for our application. The code block in Figure 5.6 contains a few of the

Figure 5.6: Transformation rules in the event scheduler.

```
rules
  addEventToEmpty (day a) =
    => a firstOf day, a eventEndOf day

  insertEventAtBeginning (day a b) = b firstOf day
    => a firstOf day, a precedes b

  addBetweenEvents (a b c) = a precedes c
    => a precedes b, b precedes c

  ...

  swapEvents (a b c d) =
    a precedes b, b precedes c, c precedes d
    => a precedes c, c precedes b, b precedes d

  ...
```

transformation rules in the application. The first transformation rule of the code block, `addEventToEmpty`, specifies how events are added to days that have no events: the transformation is defined for two placeholders `day` of component type `Day` and `a` of component type `Event`. Before the transformation, we assume no relations on `day` and `a` and, therefore, the set of relations *Pre* of the rule is empty. After the transformation, `a` should both be the first and last event of the day, as specified with the relations `a firstOf day` and `a eventEndOf day`.

The second transformation rule, `insertEventAtBeginning`, specifies how to insert an event `a` to the beginning of a day `day`, assuming that `day` starts with the event `b`. Before the transformation, `b` should be the first event of `day`. After the transformation, `a` should be the first event of `day` and `a` should precede `b`.

The third transformation rule, `addBetweenEvents`, specifies how to insert an event `b` in between two events `a` and `b`. Before the transformation, we assume that `a` precedes `c`. After the transformation, `a` should precede `b` and `b` should precede `c`. Note that `a` or `c` is possibly connected to a day, but because the transformation does not affect these connections, the transformation rule has no placeholder for the day of the events.

The last rule of the code block, `swapEvents`, specifies one of the structural changes in our application that involves swapping of events. The transformation rule `swapEvents`

swaps two adjacent events `b` and `c` where `a` initially precedes `b` and `c` initially precedes `d`. We also define other swapping transformations for events at the beginning or end of a day.

For each of the transformation rules we specify, WarmDrink generates a JavaScript function. The function generated from a rule has the same name as the rule. Therefore, using JavaScript, we can pass WarmDrink components to these functions to make structural changes to our GUI: the function will dynamically type-check the components and check that the pre-connections of the transformation initially hold. The function then performs the transformation on the components.

5.4 Summary

We have used WarmDrink to implement dynamic structural changes in a constraint system-powered GUI. The relations between the components, that is, the constraints between the variables of the components, are described in the relation specifications. The transformation rules use these relation specifications to change the structure of the components. Because we manage the lower-level connections between components, such as variable references, in the relation specifications, we can describe structural changes in our application by declaring how relations between components should change.

Chapter 6

Discussion and future work

The goal of this thesis is to create an abstraction layer over structures of components, those appearing in GUIs in particular, so that programmers can express changes to the structures without worrying about the internals of the components. We have developed a framework, WarmDrink, that enables programmers to specify transformations on structures by only describing how the relations between components in the structure change. Such specifications are declarative and ignore lower-level properties of components.

While the framework, already in its current state, allows specifying non-trivial structural changes in GUIs, we see a potential for improvements. We describe below possible directions for further research.

Modularity WarmDrink DSL programs are not encapsulated, and in a program with several WarmDrink programs, we have no control over which components are accessed from within a particular WarmDrink program. More than one WarmDrink program can describe relations that act on the same components, and therefore it is possible that the establishing or unestablishing of relations in one of the WarmDrink programs, unintentionally, change relations that are concerns of other WarmDrink programs.

Nullable placeholders As described in Section 3.2.8, in some cases a large number of rules is needed to fully specify a structural change; which rule the programmer calls depends on where in the structure the change should be made. As an example from the event scheduling program in Section 5.3, the structural change of swapping two events requires four transformation rules: the two events that are to be swapped may be the only events that day, they may be the first events that day and have succeeding events, they may be the last events that day and have preceding events, or they may have both

preceding and succeeding events. Each rule defines swapping in one of the four cases, and the only difference between the rules is whether some of the components that the rule refers to are present or not. For more complicated structures than sequences and lists, the number of rules needed to fully define a structural change can be considerably higher, because of all the cases the rules must cover. A possible solution is to explore the concept of nullable placeholders and introduce the concept to the framework. We already used this concept in Section 3.2.8, but it is not yet fully implemented in WarmDrink. A nullable placeholder in a rule is a placeholder that may or may not hold a component when the rule is applied; if the placeholder is empty in a rule application, the relations that involve the placeholder are ignored. In our example we can denote a placeholder as nullable by attaching a question mark to it. Using nullable placeholders, a swapping transformation can be defined using a single rule:

```
swap (a? b c d?) = a precedes b, b precedes c, c precedes d
  => a precedes c, c precedes b, b precedes d
```

In the case that the events b and c are the only events that day, a and c are empty. Therefore, WarmDrink will only apply the transformation by unestablishing `b precedes c` and establishing `c precedes b`. However, this does not take into account that events have connections to days, and more investigation is therefore needed to correctly express the change using nullable placeholders.

Generic rules A way to lift the abstraction level of the framework further is to introduce *generic rules*, i.e., rules parameterized on component types and relations. If transformation rules differ only by renaming the placeholders and the relations in the rule, defining one generic rule suffices. The rule could then be instantiated with different component types and relations. There are several ways generic rules can be added to a WarmDrink implementation that targets JavaScript. One way is to transpile each generic rule to a JavaScript function that, when called at runtime, decides which instantiation of the rule to use. Another approach is to generate one function for each of the instantiations of the generic rule, and then have the programmer choose the appropriate instantiation at development time. The latter approach would be a good choice in a WarmDrink implementation targeting a strongly typed language with overloading; we could generate a function for each rule instantiation and let the type system choose the correct function.

Algebraic properties Another feature we want to explore is annotating relations with algebraic properties, such as symmetry, transitivity, or reflexivity. These annotations would further allow the framework to do static analysis on transformation rules and

report to the programmers if an algebraic property of a relation is invalidated. We give a simple example of such an invalidation: a programmer annotates a relation specification `precedes` as being anti-symmetric. In the rules section, the programmer wrongly defines a transformation rule where the two relations `a precedes b` and `b precedes a` are to be established. Because `precedes` is marked as being anti-symmetric, the framework can report on the rule and inform the programmer about the error.

Data Dependency Algebras Another direction for further research is to explore whether *Data Dependency Algebras* (DDA) [5] could be applied to structural changes in GUIs. DDAs are originally designed to decouple computations from hardware, and they are useful in the domain of parallel programming: by expressing a parallel program using DDAs, its computations can run on any (distributed) hardware where there is an embedding from the DDA into the hardware. We are curious whether DDAs could be used to map the structure of a GUI's model to the widgets in its view.

Bibliography

[1] ApplyTexas: admissions and scholarships applications for Texas institutions of higher education. Accessed on 14 May 2020.
URL: `https://www.applytexas.org/adappc/html/preview20/frs_ec.html`.

[2] Stack Overflow developer survey 2019. Accessed on 10 May 2020.
URL: `https://insights.stackoverflow.com/survey/2019`.

[3] React — a JavaScript library for building user interfaces. Accessed on 2 May 2020.
URL: `https://reactjs.org/`.

[4] Skattekalkulator. Accessed on 3 May 2020.
URL: `https://skattekalkulator.app.skatteetaten.no`.

[5] Alexa Anderlik and Magne Haveraaen. On the category of data dependency algebras and embeddings. *Proceedings of the Estonian Academy of Sciences. Physics, Mathematics*, 52, 01 2003.

[6] John Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. In *Proceedings of the International Conference on Information Processing*, pages 125–131, January 1959.

[7] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend: learn how to implement a DSL with Xtext and Xtend using easy-to-understand examples and best practices.* Packt Publishing, 2016.

[8] Fabien Campagne. *The MPS language workbench.* Campagne Lab., 2016.

[9] Boris Cherny. *Programming TypeScript: making your JavaScript applications scale.* OReilly Media, Inc., 2019.

[10] F. Drewes, H.-J. Kreowski, and A. Habel. *Hyperedge Replacement Graph Grammars*, page 95–162. World Scientific Publishing Co., Inc., USA, 1997. ISBN 9810228848.

[11] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert Massow, Wilhelm Hasselbring, and Michael Hanus. Xbase: Implementing domain-specific languages for Java. *ACM SIGPLAN Notices*, 48, September 2012. doi: 10.1145/2371401.2371419.

[12] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, and Jimi Woning. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, August 2015. doi: 10.1016/j.cl.2015.08.007.

[13] 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*, OOPSLA '10, page 97–104, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450302401. doi: 10.1145/1869542.1869559.
URL: `https://doi.org/10.1145/1869542.1869559`.

[14] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. A programmable programming language. *Commun. ACM*, 61(3):62–71, February 2018. ISSN 0001-0782. doi: 10.1145/3127323.
URL: `https://doi.org/10.1145/3127323`.

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

[16] Richard C. Gronback. *Eclipse modeling project: a domain-specific language toolkit*. Addison-Wesley, 2009.

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

Computing Machinery. ISBN 9781605582672. doi: 10.1145/1449913.1449927.
URL: https://doi.org/10.1145/1449913.1449927.

[18] Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob Smith. Algorithms for user interfaces. In *Proceedings of the Eighth International Conference on Generative Programming and Component Engineering*, GPCE '09, page 147–156, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584942. doi: 10.1145/1621607.1621630.
URL: https://doi.org/10.1145/1621607.1621630.

[19] Jaakko Järvi, Gabriel Foust, and Magne Haveraaen. Specializing planners for hierarchical multi-way dataflow constraint systems. *ACM SIGPLAN Notices*, 50, 09 2014. doi: 10.1145/2658761.2658762.

[20] Lennart C.L. Kats and Eelco Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. *SIGPLAN Not.*, 45(10):444–463, October 2010. ISSN 0362-1340. doi: 10.1145/1932682.1869497.
URL: https://doi.org/10.1145/1932682.1869497.

[21] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, pages 220–242, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-69127-3.

[22] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125 – 142, 2006. ISSN 1571-0661. doi: https://doi.org/10.1016/j.entcs.2005.10.021.
URL: http://www.sciencedirect.com/science/article/pii/S1571066106001435. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005).

[23] Richard Mitchell. *Managing complexity in software engineering*. The Institution of Engineering and Technology, 1990. ISBN 978-0863411717.

[24] Terence Parr and Kathleen Fisher. LL(*): The foundation of the ANTLR parser generator. *SIGPLAN Not.*, 46(6):425–436, June 2011. ISSN 0362-1340. doi: 10.1145/1993316.1993548.
URL: https://doi.org/10.1145/1993316.1993548.

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