Journal of Computer Languages 74 (2023) 101175

Contents lists available at ScienceDirect

Journal of Computer Languages

journal homepage: www.elsevier.com/locate/cola

A domain-specific language for structure manipulation in constraint system-based GUIs

Knut Anders Stokke [a,*], Mikhail Barash [a], Jaakko Järvi [b,a]

[a] University of Bergen, Norway
[b] University of Turku, Finland

ARTICLE INFO

Keywords:
GUI programming
Separation of concerns
Rule based systems
Declarative programming
Dataflow constraint systems

ABSTRACT

A common frustration with programming Graphical User Interfaces (GUIs) is that features for manipulating structures, such as lists and trees, are limited, inconsistent, buggy, or even missing. Implementing complete and convenient sets of operations for inserting, removing, and reordering elements in such structures can be tedious and difficult: a structure that appears as one collection to the user can be implemented as several different data structures and a web of dependencies between them. Structural modifications require changes both to the GUI's model and view, and possibly extraneous bookkeeping operations, such as adding and removing event handlers.

This paper introduces a DSL that helps programmers to implement a complete set of operations to structures displayed in GUIs. The programmer specifies structures and relations between elements in the structure. Concretely, the latter are definitions of methods for establishing and unestablishing relations. Operations that manipulate structures are specified as rules that control which relations should hold before and after a rule is applied. From these specifications, our tools generate an easy-to-use API for structure manipulation. We target constraint system-based Web GUIs: the DSL generates JavaScript and relies on dataflow constraint systems for expressing dependencies between elements in GUI structures. Our DSL gives tangible representations with well-defined operations for ad-hoc and incidental GUI structures.

1. Introduction

Programming is to a large extent about manipulating data and the structures that data resides in. We routinely take advantage of well-known abstract data types (ADT) that specify operations and behavior of lists, queues, trees, graphs, and other common structures. Data manipulated through an ADT is stored in a concrete data type implemented with the ADT's interface in mind—the standard libraries of common programming languages contain many examples. Structures we encounter in programs in practice are often more ad-hoc: they may not be neatly encapsulated within the boundaries of a canned implementation of an ADT. This is particularly true in Graphical User Interface (GUI) programming.

In GUIs, the representation of a structure is typically split between a *view* and *model*, maybe several views and models. Operations that affect the structure of one of them should be reflected on the others. Further, elements in these structures are often connected to each other in different ways, e.g., to realize dataflows. Instead of well-defined data structures, programmers must manipulate *incidental* structures, which lack direct operations for doing so.

Consider a GUI for specifying a multi-city flight search, such as the one in Fig. 1. The user of such a GUI specifies a sequence of flight segments, each with a departure and arrival city and date. Though the sequence of flights has a very tangible manifestation in the GUI, its representation in code is less concrete. Presumably the GUI's model represents the sequence of flights as a linked list or an array of flight records, and the view as sibling elements in the DOM-tree. These two projections of the same structure do not live in a single data structure with predefined methods for manipulating the sequence. The code for inserting, removing, or reordering flight segments has many responsibilities, including modifying the model's flight record list, modifying the DOM-tree's widgets, adding and removing the widgets' event handlers, and updating validation logic (e.g., to ensure a chronological order when entering dates)—and to keep these changes in sync. All this is a considerable programming effort.

Features for structure modification are limited and cumbersome in many widely used GUIs. Continuing with the flight search example, almost no flight booking service allows adding flight segments in the beginning or middle of a multi-city search; we checked 30 services, one has this feature. Frequent travelers know that this would be a convenient feature when exploring options for complex itineraries, yet even services with millions of users do not provide it.

* Corresponding author.
E-mail addresses: knut.stokke@uib.no (K.A. Stokke), mikhail.barash@uib.no (M. Barash), jaakko.jarvi@utu.fi (J. Järvi).

https://doi.org/10.1016/j.cola.2022.101175
Received 14 June 2021; Received in revised form 19 June 2022; Accepted 7 November 2022
Available online 14 November 2022

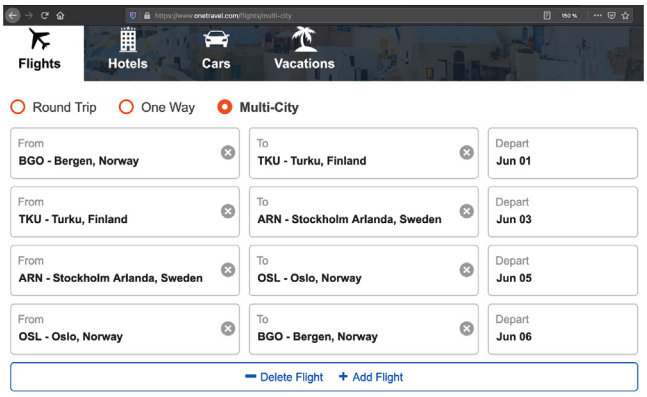

Fig. 1. A GUI for specifying a multi-city flight query. Operations for manipulating the segment sequence as a structure are rudimentary.

GUIs without adequate support for structural operations can be frustrating. In our experience, such GUIs are particularly common in in-house applications and other applications with smaller user bases, likely because in developing such applications, resources that can be invested in producing feature-rich GUIs are limited. As a representative example, the *ApplyTexas* [1] website for admissions to Texas' higher education institutions has a tab for entering extracurricular activities. The GUI asks for up to ten of them, in priority order, but offers no reordering operations. To move an activity up or down in the form would require swapping activities by swapping (with copy–paste) each of the 23 fields (text, list, and checkboxes) of two activities. In practice, filling the form again is the fastest way to perform a reordering. This time-wasting GUI is a necessary evil for hundreds of thousands of students every year, and has been for more than a decade.

It is well known that implementing GUIs constitutes a significant part of all development effort.[1] Getting developers to write GUIs with rich features requires thus either sufficient incentives or that such features are not costly to implement. The goal of this paper is to show the latter, that structural operations are implementable with little effort. Central is to make the incidental structures that appear in GUIs explicit to the programmer.

We present a domain-specific language that provides an abstract but precise view over GUIs' messy incidental structures. With this DSL, we call it *WarmDrink*, programmers define relations between elements, such as an element preceding another in a sequence, and how these relations are established and unestablished. Programmers then define an API for structural operations as transformation rules in terms of the relations: each rule defines which relations hold before and after a rule is applied. Rules can typically be parameterized over relations—a handful of relation definitions on elements of a structure that appears on a GUI thus suffices to give an API for insertions, deletions, and reorderings for that structure.

This paper builds on our prior work [4], a simpler DSL where the programmer implements relations' establishing and unestablishing code directly in JavaScript. We showed a mock implementation in that DSL of the above discussed ApplyTexas-form with a full suite of reordering operations. The DSL presented in this paper still generates JavaScript, but it describes GUI structures explicitly with grammars that specify valid compositions of structural components; these components are conceptual, and typically include elements from the GUI's view and model. A departure from [4] is that we make stronger assumptions about the structure: the DSL provides explicit support for tree structures and offers XPath-like tree-navigation expressions for such structures. This navigation capability simplifies the implementation of structure transformation rules, and in particular rules parameterized over relations, which we now support (as alluded to in the description of future work in [4]).

WarmDrink is the result of exploring and "teasing out" the structural similarities found in different aspects of a GUI, in the view and the model. The paper shows that by making the similarities visible by binding these aspects to one explicit structure, providing a unified navigation syntax for these different aspects, and making the structure accessible from any view object, a great deal of structure manipulation code becomes easily reusable.

2. Baseline: structures in contemporary GUI programming

GUI frameworks let programmers write *handler functions* that respond to user events, such as clicking a button, dragging a slider, or typing in a text field. It is well-known that unstructured event-handling code easily becomes interdependent "spaghetti" that is prone to defects and difficult to comprehend and maintain [5]. To manage the complexity of GUI code, many software patterns and architectures (e.g., MVC [6], MVVM [7], MVP [8], and MVU, also known as the Elm Architecture [9]) have been developed. GUI languages and libraries that realize and support variations of these patterns and architectures abound; some of the recent ones include Elm, Vue, Angular, React, Knockout, and ReactiveUI.

As described in the introduction, the state of the GUI has its manifestations both in the (view)model and in the view. A common goal of all the above patterns is to ensure a single source of truth of the GUI's state, and to keep the view devoid of logic. In all these patterns, user events from view elements (widgets) are interpreted as requests to modify the model, and once the model is updated accordingly, the view is adjusted to reflect the new state of the model.

The connection between the view and the model is particularly explicit in the MVVM pattern through its *data binders* concept (see, e.g., the KnockoutJS library [10]). A binder connects an element in a view with a particular piece of data in the view model, typically via a two-way observer-observable connection. Binders guarantee that the view stays in sync with the view-model and vice versa. The MVVM pattern, as well as MVC and MVP, are however silent on how to manage changes of the structure of the view model and view, e.g., how to ensure that bindings are created or disconnected when new view and model elements are added, removed, or reorganized.

Elm, that follows the MVU pattern, and React [11] use a different approach: the view in its entirety, both the structure and content, is derived from the model. Modifications on the structure of the model are then automatically also modifications on the structure of the view because the view is re-rendered when the structure of the model changes. To keep the approach efficient, e.g., React first writes its tree to an internal data structure (*Virtual DOM*) and then *reconciles* [12] it with the prior state of the browser's DOM.

Our work focuses on GUI programming according to the MVVM pattern. The motivation comes from cases where the view-model itself has a complex structure, when there are dependencies between components of the view-model. Then, structural modifications can be complex even if the model completely determines the view. Complex dependencies within the view-model arise, e.g., when programming GUIs using reactive programming (see, e.g [13]). Central to reactive programming is an explicit specification of dataflow: programmers define dependencies between data streams, typically as methods that react to changes in their input data streams and compute new values to their output data streams. Then, when a structure of a reactive view-model changes, the programmer has to ensure that new reactive programs, new dataflows, are correctly established in the changed view-model.

The DSL introduced in this paper has explicit support for updating dataflows in the view-model. Concretely, the DSL interfaces with the "HotDrink" [14–16] library, which makes defining dataflows particularly easy.

[1] Myers' early study [2] reports about 50% of all code to be related to user interfaces, Parent puts that number to 30% of all Adobe applications code and remarks that this code has a disproportionately higher share of defects [3].

2.1. Constraint systems for GUIs

The HotDrink library is based on *multiway dataflow constraint systems* [17]. A dataflow constraint defines a relation over a set of variables as a set of methods, each of which can enforce the constraint, i.e., compute a variable valuation that satisfies the relation. For example, the following HotDrink program specifies a system of four variables and two constraints.

```
const Rectangle = hdl`
 component {
  var A, w, h, p;
  constraint {
   m1(w, h -> p) => 2*(w+h);
   m2(p, w -> h) => p/2 - w;
   m3(p, h -> w) => p/2 - h;
  }
  constraint {
   n1(w, h -> A) => w*h;
   n2(A -> w, h) => [Math.sqrt(A), Math.sqrt(A)];
  }
 }`;
```

The first constraint is between the perimeter p, width w, and height h of a rectangle, and the second between w, h, and the area A. The first constraint specifies methods from any of its two variables to the third one. The second constraint specifies two methods, from w and h to A and from A to w and h (in the last case the programmer has chosen to default to a square). The method bodies can be arbitrary JavaScript code. A method that has more than one output returns its results as an array.

Concretely, HotDrink programs are embedded to JavaScript as tagged template literals. The result of the `hdl`-tagged template literal is a *component* of variables and constraints. Components can also be constructed without the DSL, less conveniently, using an API (not shown here), or by cloning existing components.

Every time any of a constraint system's variables is assigned to, a constraint solver determines which methods to execute in which order to enforce so that all constraints become enforced. When specifying dependencies as multiway dataflow constraint systems, a programmer essentially describes many reactive programs, or many dataflows at once, of which the constraint solver chooses the most appropriate one in each state.

Constraint systems have been studied extensively in the context of user interfaces and a large number of declarative, constraint-based GUI systems have been proposed, including Sketchpad [18], Amulet [19], Garnet [20], and ThingLab I and II, DeltaBlue, and SkyBlue [21]. The applications of these (now old) systems were mostly for expressing geometric constraints, e.g., for automatic widget layout. A modern realization of a constraint-based layout is Apple's Auto Layout [22].

In HotDrink GUIs, the use of constraint systems extends beyond layout: the view-model is a constraint system. The programmer binds widgets to the constraint systems' variables, user events on widgets inform the system that a variable's value has changed, which triggers a constraint solver to produce a new variable valuation and update views through bindings. We have shown that using constraint-systems as view-models allows for implementing several GUI features as reusable algorithms [15,23,24]. E.g., HotDrink knows at all times which variables have pending values (so that widgets bound to them can show an indicator) or which variables are irrelevant in the current dataflow (so that widgets bound to them can be disabled automatically).

HotDrink allows for building constraint systems piecemeal from components. Some of the variables of a component can be *references*, which can be bound to variables of other components to form constraints across components. This binding is accomplished simply by assigning a variable to a reference. Connecting and disconnecting HotDrink components is then the low-level plumbing that WarmDrink builds upon: structural modifications are realized by connecting and disconnecting HotDrink components—and making the corresponding view changes. To keep the terminology clear, below we refer to HotDrink's constraint system components as *cs-components*.

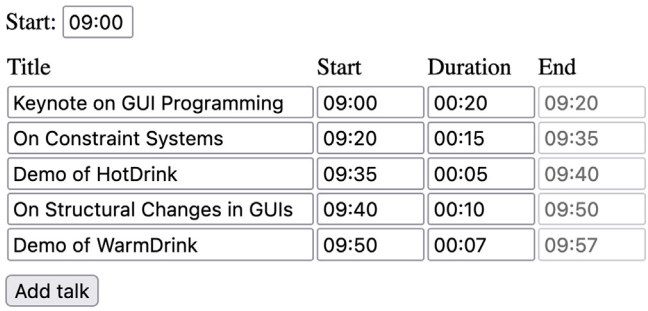

Fig. 2. A GUI for planning the schedule for a sequence of events.

```
component Agenda {
 var start="09:00", duration="00:00";
}
component Talk {
 var title, start, duration, end,
     &prevStart, &prevDuration;
 constraint EndIsStartPlusDuration {
  (start, duration -> end) => addTimes(start, duration);
 }
 constraint AdjacentTalks {
  (prevStart, prevDuration -> start) =>
   addTimes(prevStart, prevDuration);
 }
}
```

Listing 1: The constraint system specification for the conference scheduler application.

2.2. Running example: a conference day

As an example of a user interface with connected cs-components, consider the application in Fig. 2 for scheduling events on an agenda, e.g., talks at a conference. For each talk the GUI shows the *title*, *start time*, *duration*, and *end time*. This last data item is the combined duration of the current talk and all prior ones.

As seen in Listing 1, the constraint system underlying the GUI comprises one `Agenda` and many `Talk` cs-component instances. The value of `Agenda`'s variable `start` determines when a conference day begins. `Talk`'s three variables `start`, `duration` and `end` store, respectively, the current talk's start time, duration, and end time. The references `prevStart` and `prevDuration` connect two talks; when appending a new talk to a sequence of talks in JavaScript, the programmer assigns variables `start` and `duration` of the preceding talk to `prevStart` and `prevDuration`, respectively, which enables the component to use information from the previous talk when computing its own variables. Thus, each talk in the sequence stores two references to its preceding talk. The first talk connects the two references differently, to an `Agenda` component's variables `start` and `duration`. The latter variable is never updated nor shown in the GUI, but it is needed by the first talk of the sequence.

`Talk`'s `EndIsStartPlusDuration` constraint maintains the ternary relation on the current talk's start time, duration and end time: the method[2] that writes a new value to its output `end` executes

[2] This simple constraint has only one method; in general a constraint can have several methods, each defining a different dataflow.

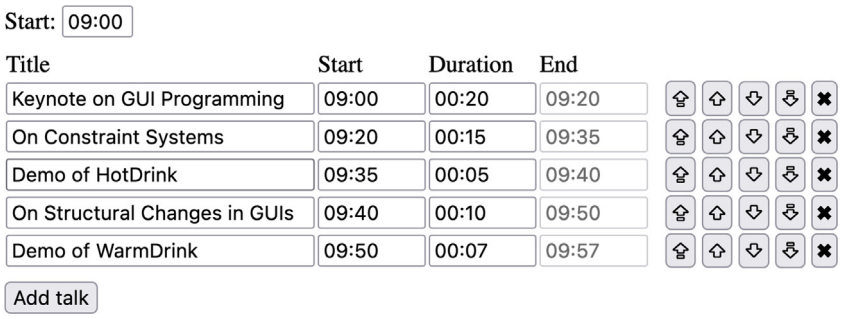

Fig. 3. The GUI from Fig. 2 augmented with proper structure manipulation features.

when either of its inputs, `start` or `duration`, changes. The second constraint, `AdjacentTalks`, ensures that each talk starts when the previous talk ends.

To add a new talk to the agenda is not entirely trivial. We do not show the code, but describe the main points. One must first create an instance of the `Talk` cs-component, and connect it to the previous `Talk` instance. The view visible to the user must also be extended. This means creating a few new DOM elements to display the cs-component instance's content:

```
<li>
  <input data-bind="title" />
  <input data-bind="start" />
  <input data-bind="duration" />
  <input data-bind="end" disabled />
</li>
```

In this code the `data-bind` attributes specify that a view element is bound to a particular variable in a cs-component; HotDrink can inspect these attributes and realize these bindings.

2.3. Manipulating agendas

The GUI in Fig. 2 supports only one simple structural operation: adding a new talk. A user-friendly GUI needs many more: Fig. 3 adds buttons for moving talks up or down, making them the first or last, and deleting them. Implementing such removal and reordering operations involves both updating the view and the connections between the cs-components in the constraint system. For example, removing a talk means disconnecting it from its predecessor and successor, then connecting the successor to the predecessor. Removing the first talk in an agenda is a special case, since it has no predecessor but is connected to the agenda's two variables. If cs-components are stored in some container in the model, that container must be updated; in the conference planner described in Section 4.4, for example, agendas are nested in another structure. The view must be modified to reflect the new state, some DOM-nodes must be removed, and if the GUI is connected to a backend server, also the data on the server need to be updated.

It is clear that the agenda structure is notably more complex than a container object with predefined operations for addition, deletion, and reordering. Different aspects of this incidental structure appear in the view, model, and backend server. It falls on the programmer to ensure that these different aspects stay in sync with each change to the agenda's structure.

3. WarmDrink: a DSL for structure manipulation

The goal of our WarmDrink DSL is to relieve the programmer from the kind of tedious low-level programming of structural operations described in the previous section. Below we use WarmDrink to extend the conference day application with functionality to swap two adjacent talks, to move a talk to the beginning or the end of the agenda, and to remove a talk from the agenda. These structural manipulations should perform corresponding updates of the connections between the talks and the agenda in the constraint system.

A WarmDrink specification is a tuple $\langle \mathbf{S}, \mathbf{F}^{\mathtt{js}}, \mathbf{R}, \mathbf{T} \rangle$, where:

- $\mathbf{S}$ is a *structure* specification,
- $\mathbf{F}^{\mathtt{js}}$ is a set of *subroutines* defined on elements of the structure,
- $\mathbf{R}$ is a set of finitary *relations* defined on elements of the structure,
- $\mathbf{T}$ is a set of *transformation rules* that express (all) possible manipulations that modify component relations.

The structure specification defines *semantically meaningful* components of a GUI, which are called *wd-component*s in what follows. Syntactically, a declaration of a wd-component consists of the component name, bindings to a constraint system component (which define some of cs-component's variables and references as connectors: a reference of one component can be bound to a variable of another), and nested wd-components (within curly braces). Nested components have a *cardinality*, which can either be 1 or $0..n$ (denoted by $*$). The full grammar of the WarmDrink specification language is given in Appendix.

Formally, a GUI structure $S = (V, E)$ is a rooted directed acyclic graph where (1) V is a set of labeled vertices that represent wd-components; (2) $v_{\text{root}} \in V$ is a designated root vertex; and (3) E is a set of labeled edges with labels of the form f or f^* that represent *features* inside structural elements and define the cardinality of those features (no superscript means one, $*$ means zero or more). An edge (v_A, v_B) with label f means that the wd-component v_B is nested in the wd-component v_A, and v_A is said to be the *parent* of v_B.

Each subroutine in $\mathbf{F}^{\mathtt{js}}$ is either a predicate or a procedure, and its body contains blocks of imperative JavaScript code.

A relation r in $\mathbf{R}$ is defined on instances of wd-components and is a triple $\langle r^{\mathtt{test}}, r^{\mathtt{establish}}, r^{\mathtt{unestablish}} \rangle$, where:

- $r^{\mathtt{test}}$ is a call to a predicate from $\mathbf{F}^{\mathtt{js}}$ that *tests* whether the relation holds;
- $r^{\mathtt{establish}}$ is an optional code block that *establishes* the relation;
- $r^{\mathtt{unestablish}}$ is an optional code block that *unestablishes* the relation.

Syntactically a relation specification is a name, a list of arguments in parentheses, and three code blocks (`test`, `establish`, and `unestablish`) enclosed in curly braces. An (imperative) code block is a sequence of statements, where each statement is a *function call* to a subroutine in $\mathbf{F}^{\mathtt{js}}$ or a *reference update* of the form `a.x = b.y`; its meaning is to bind the reference `x` in wd-component `a`'s model to the variable `y` in wd-component `b`'s model.

Transformation rules perform structural modifications and are defined in terms of relations. A transformation rule t in $\mathbf{T}$ is a tuple $\langle t^{\text{premises}}, t^{\text{conseq}} \rangle$, where t^{premises} is a sequence of *premises*, relations that must hold for the rule to be applicable, and t^{conseq} is a sequence of *consequences*, relations that shall hold after the rule has been applied.

```
<body>
 <div ...>
  <ul>
   <li>
    <input data-bind="title" />
    <input data-bind="start" />
    <input data-bind="duration" />
    <input data-bind="end" disabled />
   </li>
   <li>
    <input data-bind="title" />
    <input data-bind="start" />
    <input data-bind="duration" />
    <input data-bind="end" disabled />
   </li>
   ...
  </ul>
  <button id="add-new" onclick="addTalk()">Add talk
  </button>
 </div>
</body>
```

Listing 2: An HTML snippet from our example GUI.

An application of a transformation rule t unestablishes all relations from t^{premises}, establishes all relations from t^{conseq}, and tests that they hold. Syntactically, a rule specification is a name, followed by a list of arguments in parentheses and the rule's body enclosed in curly braces. The body is a (possibly empty) list of premises, an arrow sign, and a (possibly empty) list of consequences.

3.1. Running example: specifying structures

The first step with a WarmDrink specification is to define the structure of a GUI. The DOM reflects this structure to an extent; in our example GUI, an agenda and talks form a fraction of a tree, which is reflected in its HTML representation, as shown in Listing 2. The DOM and its HTML, however, contains clutter: layout- and styling-specific tags and tags that do not map to *semantically meaningful* components of the GUI. WarmDrink's `structure` declaration below expresses the structure without clutter: the conference day application contains an agenda, which is a list of talks.

```
structure
 root ConferenceApp {
  agenda: Agenda
 }
 Agenda [[ var start, var duration ]] {
  talks: Talk*
 }
 Talk [[ var start, var duration, var end, ref prevStart,
        ref prevDuration ]]
```

We call the elements `ConferenceApp`, `Agenda`, and `Talk` in this declaration *wd-components*.

`ConferenceApp`'s *feature* `agenda` defines a nested wd-component `Agenda` which has cardinality 1. Component `Talk` is nested in `Agenda`, and it has cardinality $0..n$, which is denoted by the star ($*$) to the right of `Talk`.

A wd-component stores all information related to the structural element it represents. E.g., a wd-component of a talk holds references to the view, the part of the DOM-tree that displays the talk's information, and to the model, a cs-component storing all its data. The programmer does not have to hold on to wd-components: one can access the relevant wd-component from any DOM-element that is part of a semantic component.

The bindings, variable names in double square brackets, define some of cs-component's variables and references as connectors: a reference of one component can be bound to a variable of another.

3.2. Running example: the populated structure at run time

When the application is running, the GUI structure S is populated with *instances* of wd-components. This populated structure $\widehat{S}$ is a tree, where each node $\hat{s} \in \widehat{S}$ stores a reference to the instance of its cs-component and the corresponding DOM-node, which we, respectively, denote by $\mathsf{Model}(\hat{s})$ and $\mathsf{View}(\hat{s})$.

The realization of $\widehat{S}$ is a JavaScript object that follows the structure S: its keys correspond to features that refer to nested wd-components (cardinality 1) or nested arrays of wd-components (cardinality $*$). This object that represents the conceptual – or semantic – structure holds references to the bits of the view (fragments of the DOM) and the model (constraint system components) that constitute the entire concretization of the structure. When the structure is manipulated by the rules of the DSL, changes to the conceptual structure effect the appropriate changes to the concretization.

Fig. 4 shows a schematic of a populated structure of the conference day GUI. The $\widehat{\texttt{Agenda}}$ node represents the structural element `Agenda` and stores a view reference to the corresponding `div`-node in the DOM. `Agenda`'s `talks` feature is a subtree $\widehat{\texttt{Talk}\,*}$, whose view reference is to the `ul`-node. The child nodes' view references are to `li`-nodes and model references to instances of `Talk` cs-components. All the view references are bidirectional, so that one can get to wd-components from the DOM.

Formally, given a structure specification $S = (V, E)$, the *populated structure* $\widehat{S} = (\widehat{V}, \widehat{E})$ is defined as follows: (1) for vertex v_{root} in V, there is a vertex $\widehat{v_{\text{root}}}$ with label `root` in $\widehat{V}$; (2) for an edge $(v_A, v_B) \in E$ labeled f with cardinality 1, there is an edge $(\widehat{v_A}, \widehat{v_B}) \in \widehat{E}$ labeled f; and (3) for an edge $(v_A, v_B) \in E$ labeled f^*, there is an edge $(\widehat{v_A}, \widehat{v_{f*}}) \in \widehat{E}$ labeled f, and an edge $(\widehat{v_{f*}}, \hat{s})$ for each instance $\hat{s}$ of wd-component v_B. The node $\widehat{v_{f*}}$ is a *container* node, which is distinct from an instance's parent node for features with cardinality $*$.

3.3. Running example: manipulating the populated structure

Listing 3 shows the complete WarmDrink program that generates an API for swapping talks in the conference day GUI. It is in many ways limited, yet sufficient for an overview of different parts of a WarmDrink program. The sections that follow give further details.

Line 2 in Listing 3 imports the functions `hasNext` and `insertAfter` used in the program; Section 4.1 discusses their implementation. WarmDrink's simple module system is explained in Section 5.4.

Lines 5–8 repeat the structure specification discussed above. Swapping is based on the binary *relation* `precedes`, on line 9, that expresses the fact that a talk instance $\hat{s}_a$ precedes another instance $\hat{s}_b$ in a given agenda in the populated structure. A relation has *test* code for checking if the relation holds (line 10). It also has *establish* code for making the relation hold: line 13 inserts the instance $\hat{s}_b$ after the instance $\hat{s}_a$ in the populated structure. The `hasNext` and `insertAfter` functions operate on instances of wd-component `Talk`. Lines 15–15 define the connections between the cs-components $\mathsf{Model}(\hat{s}_b)$ and $\mathsf{Model}(\hat{s}_a)$ when they are adjacent.

The *transformation rule* `swapBetween` in lines 20–23 defines swapping of two adjacent talks `b` and `c`, surrounded by talks `a` and `d`. This context is needed so that the swapped elements' cs-components' connections to `a` and `d` are correctly updated. The *premises* (line 20) of this rule specify what should hold before the rule is applied: `precedes` for the pairs $(\hat{s}_a, \hat{s}_b)$, $(\hat{s}_b, \hat{s}_c)$, and $(\hat{s}_c, \hat{s}_d)$ of `Talk` instances. The *consequences* (line 22) specify what should hold after the rule is

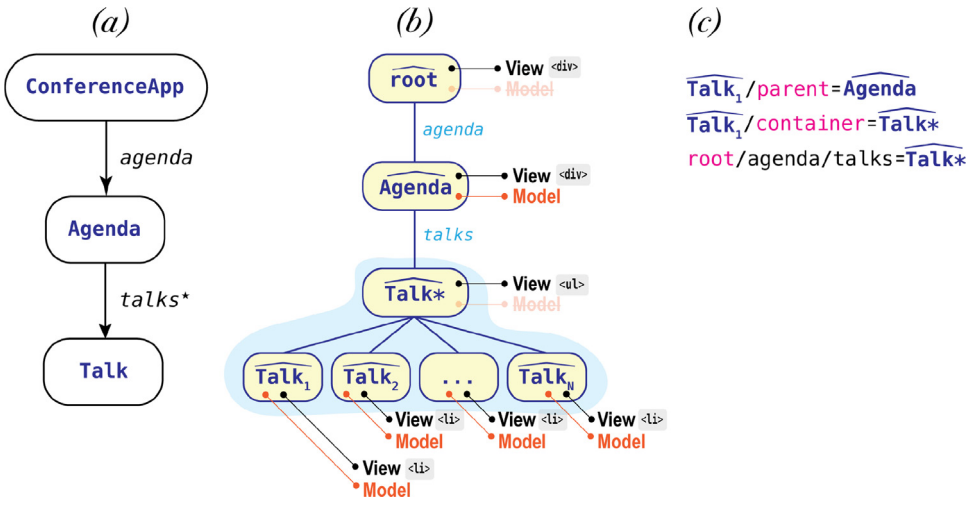

Fig. 4. (*a*), (*b*) visual representation of the structure S and a populated structure $\hat{S}$ of the conference day GUI; (*c*) XPath-like expressions used to navigate the populated structure. Note the difference between the parent and the container of $\widehat{\texttt{Talk}_1}$.

```
1  module org.example.conference
2  import warmdrink.library.lists.*  // imports hasNext
       and insertAfter
3
4  structure
5   root ConferenceApp {  agenda: Agenda }
6   Agenda [[ var start, var duration ]] { talks: Talk* }
7   Talk [[ var start, var duration, var end,
8          ref prevStart, ref prevDuration ]] relations
9   precedes(Talk a, Talk b) {    test {
10    hasNext(a, b)       // relation holds if the next
        element of a is b
11   }
12   establish {
13    insertAfter(b, a); // put b after a     // update
        constraint system's dependencies
14
15    b.prevStart = a.start;     b.prevDuration = a.
        duration;   }
16  }
17  // other relations ...
18
19 rules
20  swapBetween(Talk a, Talk b, Talk c, Talk d) {    a
       precedes b, b precedes c, c predeces d =>
21    // given order a, b, c, d
22   a precedes c, c predeces b, b precedes d     //
       establish a, c, b, d
23  }   // other rules ...
```

Listing 3: The outline of WarmDrink code for structural manipulation of the conference day GUI.

applied: `precedes` for the pairs $(\hat{s}_a, \hat{s}_c)$, $(\hat{s}_c, \hat{s}_b)$, and $(\hat{s}_b, \hat{s}_d)$. That is, the instances $\hat{s}_b$ and $\hat{s}_c$ should be swapped.

The code generated from this rule *tests* the premises, *unestablishes* them (nothing to do in this example) and then *establishes* the consequences, by running the relations' `establish` code. It swaps the instances $\hat{s}_b$ and $\hat{s}_c$ in the populated structure, and updates the constraint system dependencies to the new order. The swapping here is limited to cases where neither the first nor the last element is swapped.

```
functions
 bool hasNext(any a, any b) {
  structure { js'''«a/following-sibling» === «b»''' }
 }
 void insertAfter(any b, any a) {
  structure { js'''
    const indexX = «a/container».indexOf(«a»)
    const indexY = «a/container».indexOf(«b»)
    if (indexY !== -1) «a/container».splice(indexY, 1)
    «a/container».splice(indexX + 1, 0, «b»)
  ''' }
  view { js'''«a/container».insertBefore(«b», «a/
    following-sibling»)
  ''' }
 }
```

Listing 4: The definitions of functions `hasNext` and `insertAfter`.

Section 4.3 elaborates on how to implement general swapping that does not have this limitation.

4. Defining relations and transformation rules

This section describes WarmDrink's basic features in detail.

4.1. Defining relations

WarmDrink supports relations of arbitrary arity and argument types. Relations are defined using functional notation, and their uses assume either prefix (for unary relations) or infix (for relations with arity two or higher) notation.

As described in Section 3, a relation defines the code blocks for testing, establishing, and unestablishing the relation. These imperative code blocks are where the real work happens. For the most part, the functions that are called from those blocks are reusable and can be considered to be part of WarmDrink's "standard library", but an application programmer might write them too. We show the definitions of `hasNext` and `insertAfter` in Listing 4.

Functions are either predicates (return `bool`) or procedures (return `void`). Their body is split into two *concerns*, `structure` and `view`, which represent the changes to, respectively, nodes in the populated structure and their views (DOM nodes). These blocks are JavaScript

```
relations
 ...
 isFirstTalkIn(Talk t, Talk* talks) {
  test {
   isFirstInItsContainer(t)
  }
  establish {
   insertAtBeginning(talks, t);
   t.prevStart = |t/parent|.start;
   t.prevDuration = |t/parent|.duration;
  }
 }
 isLastTalk(Talk t) {
  test {
   isLastInItsContainer(t)
  }
 }
```

Listing 5: The definitions of relations `isFirstTalkIn` and `isLastTalk`.

```
functions
 ...
 bool isFirstInItsContainer(any x) {
  structure { js'''
   «x/container/first-child» === «x»
  '''}
 }
 void insertAtBeginning(any cont, any x) {
  structure { js'''
   const indexX = «cont».indexOf(«x»)
   if (indexX !== -1) «cont».splice(indexX, 1)
   «cont».splice(0, 0, «x»)
  '''}
  view { js'''
   «cont».insertBefore(«x», «cont/first-child»)
  '''}
 }
 bool isLastInItsContainer(any x) {
  structure { js'''
   «x/container/last-child» === «x»
  '''}
 void insertAtEnd(any cont, any x) { /* ... */ }
```

Listing 6: The definitions of the functions used in relations `isFirstTalkIn` and `isLastTalk`.

code, enclosed in triple quotation marks; the JavaScript code can be spliced with WarmDrink code, as explained below.

The `hasNext` predicate is invoked from the `test`-block of `precedes`, and possibly other relations. It queries the populated structure and the view to confirm that `b` follows `a`, as expected. The `insertAfter` procedure modifies the populated structure, which requires some messy array manipulation. The corresponding manipulation of the DOM is simple; the assumption here is that the view consists of consecutive DOM-elements. Would this not hold, the application programmer would write a different procedure.

To facilitate navigation within the populated structure, we introduce *x-expressions*, similarly to how XPath expressions are used to navigate XML trees [25]. From within a function's JavaScript code, one can splice x-expressions using guillemets (« and »), similar to template strings in Eclipse Xpand [26]. As in XPath expressions, an x-expression is a sequence of *steps* separated by "/". The first step is a component instance, which can be either the root element (`root`) of the structure, or an argument (e.g., `a`). Each subsequent step is either an *axis* specifier (one of `parent`, `container`, `preceding-sibling`, `following-sibling`, `first-child`, `last-child`) or a feature in the structure $\hat{\mathcal{S}}$ (e.g., `agenda`, `talks`). The `parent` axis is defined for all nodes in $\hat{\mathcal{S}}$ but the root; `container`, `preceding-sibling` and `following-sibling` are defined for nodes that are list elements; and `first-child` and `last-child` are defined for nodes that are lists. Examples of x-expressions are given in Fig. 4(*c*).

A spliced x-expression x that appears within a `structure`-block of a function refers to a node $\hat{s} \in \hat{\mathcal{S}}$, whereas x appearing within a `view` block refers to $\mathsf{View}(\hat{s})$.

Accessing elements with x-expressions simplify relation definitions. The `precedes` relation in Listing 3, for example, is defined simply on two consecutive `Talk` instances, without mentioning the container they reside in. This is possible, because the container can be accessed from any of its elements: if `a` is an element, `a/container` is the container.

In Listing 5, we present two more relations, `isFirstTalkIn` and `isLastTalk`, to showcase different approaches to access nodes of the populated structure. These relations, as well as the relation `precedes`, are used in transformation rules that implement swapping, inserting, and removing talks in an agenda. The `isFirstTalkIn` relation is defined on a talk and a list of talks. This relation expresses the fact that `a` is the first child of `talks`, and it is established by inserting `a` at beginning of `talks`, both in the populated structure and the view. The `isLastTalk` relation is analogous, but for the last element. The `unestablish`-blocks are not necessary on either relation, and since our program only uses `isLastTalk` as a premise in a transformation rule, it does not need an `establish`-block.

The above two relations rely on four new functions, defined in Listing 6. Note that the two predicates `isFirstInItsContainer` and `isLastInItsContainer` operates only on the populated structure; one can assume that whenever the predicates hold for the structure, they also hold for the view.

4.2. Defining transformation rules

Section 3.3 discussed briefly the transformation rule for swapping talks (lines 20–23 in Listing 3) and explained how the rule "executes" by running the `unestablish` codes of the premises and `establish` codes of the consequences. The premise and consequence lists define the order in which relations are unestablished and established; the order may matter since the (un)establishing code is imperative.

We now explain in more details how the models' dependencies between cs-components get updated based on the two assignments in the `precedes` relation in line 15–15 (of Listing 3). This specification states that for every pair $(\hat{s}_i, \hat{s}_{i+1})$ of adjacent instances of wd-component `Talk`, the references `prevStart` and `prevDuration` of $\mathsf{Model}(\hat{s}_{i+1})$ should respectively point to the variables `start` and `duration` of $\mathsf{Model}(\hat{s}_i)$. The rule involves altogether four `Talk` arguments `a`, `b`, `c`, and `d`, of which it swaps the middle two. Hence, some consecutive instances $\hat{s}_\mathtt{a}$, $\hat{s}_\mathtt{b}$, $\hat{s}_\mathtt{c}$ and $\hat{s}_\mathtt{d}$ end up in order $\hat{s}_\mathtt{a}$, $\hat{s}_\mathtt{c}$, $\hat{s}_\mathtt{b}$ and $\hat{s}_\mathtt{d}$. All old connected pairs of instances should be disconnected, and the new pairs connected. That is, before the transformation rule is applied, the components of the constraint system are connected as follows:

$$\mathsf{Model}(\hat{s}_\mathtt{b}).\mathtt{prevStart} \equiv \mathsf{Model}(\hat{s}_\mathtt{a}).\mathtt{start}$$
$$\mathsf{Model}(\hat{s}_\mathtt{b}).\mathtt{prevDuration} \equiv \mathsf{Model}(\hat{s}_\mathtt{a}).\mathtt{duration}$$
$$\mathsf{Model}(\hat{s}_\mathtt{c}).\mathtt{prevStart} \equiv \mathsf{Model}(\hat{s}_\mathtt{b}).\mathtt{start}$$
$$\mathsf{Model}(\hat{s}_\mathtt{c}).\mathtt{prevDuration} \equiv \mathsf{Model}(\hat{s}_\mathtt{b}).\mathtt{duration}$$
$$\mathsf{Model}(\hat{s}_\mathtt{d}).\mathtt{prevStart} \equiv \mathsf{Model}(\hat{s}_\mathtt{c}).\mathtt{start}$$
$$\mathsf{Model}(\hat{s}_\mathtt{d}).\mathtt{prevDuration} \equiv \mathsf{Model}(\hat{s}_\mathtt{c}).\mathtt{duration}$$

```
swapAtBeginning(Talk b, Talk c, Talk d, Talk* talks) {
 b isFirstTalkIn talks, b precedes c, c precedes d =>
 c isFirstTalkIn talks, c precedes b, b precedes d
}
swapAtEnd(Talk a, Talk b, Talk c, Talk* talks) {
 a precedes b, b precedes c, isLastTalk c =>
 a precedes c, c precedes b, isLastTalk b
}
swapWhenOnlyTwo(Talk b, Talk c, Talk* talks) {
 b isFirstTalkIn talks, b precedes c, isLastTalk c =>
 c isFirstTalkIn talks, c precedes b, isLastTalk b
}
```

Listing 7: The definitions of transformation rules `swapAtBeginning`, `swapAtEnd`, and `swapWhenOnlyTwo`.

```
swap(Talk a = b/preceding-sibling, Talk b, Talk c,
  Talk d = c/following-sibling, Talk* talks = b/container)
{
 case a precedes b, b precedes c, c precedes d =>
      a precedes c, c precedes b, b precedes d

 case b isFirstTalkIn talks, b precedes c, c precedes d =>
      c isFirstTalkIn talks, c precedes b, b precedes d

 case a precedes b, b precedes c, isLastTalk c =>
      a precedes c, c precedes b, isLastTalk b

 case b isFirstTalkIn talks, b precedes c, isLastTalk c =>
      c isFirstTalkIn talks, c precedes b, isLastTalk b
}
```

Listing 8: The definition of the multi-case transformation rule `swap`.

Unestablishing all three `precedes` relations in the rule's premises disconnects these connections, and establishing the three `precedes` relations in consequences reconnects the cs-components as follows.

$$\text{Model}(\hat{s}_c).\text{prevStart} \equiv \text{Model}(\hat{s}_a).\text{start}$$
$$\text{Model}(\hat{s}_c).\text{prevDuration} \equiv \text{Model}(\hat{s}_a).\text{duration}$$
$$\text{Model}(\hat{s}_b).\text{prevStart} \equiv \text{Model}(\hat{s}_c).\text{start}$$
$$\text{Model}(\hat{s}_b).\text{prevDuration} \equiv \text{Model}(\hat{s}_c).\text{duration}$$
$$\text{Model}(\hat{s}_d).\text{prevStart} \equiv \text{Model}(\hat{s}_b).\text{start}$$
$$\text{Model}(\hat{s}_d).\text{prevDuration} \equiv \text{Model}(\hat{s}_b).\text{duration}$$

This explains why swapping needs to be defined in terms of four elements.

4.3. Defining multi-case transformation rules

The `swapBetween` rule in Listing 3 generates code for swapping `Talk` instances $\hat{s}_b$ and $\hat{s}_c$, as long as the surrounding instances $\hat{s}_a$ and $\hat{s}_d$ are present in the container. The rule cannot thus be applied if $\hat{s}_b$ is the first or $\hat{s}_c$ the last element of the list, or both. We need the three additional rules shown in Listing 7 to cover the missing cases. In total, four different rules are needed because the handling of connections is different in each case: swapping the first element means connecting its references to variables in the `Agenda` component, not to another `Talk`, for example.

Four separate rules is a workable solution: for each rule one JavaScript function is generated and the application programmer makes sure to invoke the right one in each of the four cases. For example, to swap two wd-components `a` and `b` of type `Talk` at the beginning of a list `talks`, the programmer invokes `swapAtBeginning(a, b, talks)`.

The rules are, however, very similar. They all swap the middle two talks `b` and `c`, but differ in whether or not there is an element that precedes `b` or follows `c`. Similarly, to support insertion of elements at the beginning of a list, in the middle, and at the end, one would have to write several rules, all with the same purpose of inserting an element. For such situations, WarmDrink provides *multi-case* rules that reuse the commonalities of several rules. The transformation rules `swapBetween`, `swapAtBeginning`, `swapAtEnd`, and `swapWhenOnlyTwo` can be combined into one *multi-case transformation rule*, as shown in Listing 8. Like the original rules, this new rule swaps component instances represented by arguments `b` and `c`. Additionally, the arguments `a`, `d`, and `talks` are defined as *implicit arguments* of the rule[3]; they can be computed from `b` and `c` using x-expressions. When using the function generated from a multi-case rule, the application programmer can invoke the transformation function with only the *explicit arguments* of the rule, in this case the two instances of wd-component `Talk` that are to be swapped. The application programmer does not have to write a complicated if-else statement (we give an example of this in Section 6) when responding to swap events; WarmDrink figures out the correct case to apply.

The machinery works as follows. First, the rule computes instances of the implicit arguments in the current populated structure. In cases where the instance of an implicit argument is missing, such as "the first child in an empty list", the implicit argument becomes null; `test`-blocks of relations that refer to a null value will always return false. After the implicit arguments are computed, each subrule is tried in their declaration order. The first subrule for which all premises hold is applied. If no such rule exists, an error is reported.

The multi-case rule `swap`, when applied to instances $\hat{s}_b$ and $\hat{s}_c$, uses implicit arguments to compute three additional instances: $\hat{s}_a = \hat{s}_b/\texttt{preceding-sibling}$, $\hat{s}_d = \hat{s}_c/\texttt{following-sibling}$, and $\hat{s}_{\texttt{talks}} = \hat{s}_b/\texttt{container}$. The first subrule applies if all instances $\hat{s}_a$, $\hat{s}_b$, $\hat{s}_c$, and $\hat{s}_d$ are present in the populated structure. The second applies if $\hat{s}_b$ is the first element in the container; $\hat{s}_a$ is then null. The third applies if $\hat{s}_c$ is the last element in the container and $\hat{s}_b$ has a previous element. Finally, if $\hat{s}_b$ and $\hat{s}_c$ are the only elements in the list, the fourth subrule is applied; $\hat{s}_a$ and $\hat{s}_d$ are then null.

4.4. Defining parameterized rules

The transformation rule for swapping is written for particular wd-component types and relations for those types. A closer inspection reveals that the rule does not rely on specific properties of those types, or relations. A swap rule for a sequence of any kind of components, with any kinds of connections between them, would be essentially the same rule. This is where *parameterized rules*, transformation rules parameterized over component types and relations, come in. Swapping and other such common structural operations can be defined using parameterized rules in a standard library, to be reused with different, but structurally similar, GUIs.

We explain parameterized rules by extending the conference day planner to a full conference planner that lets the user plan several days, each of which contains its own agenda. Fig. 5 shows a snapshot of the application's GUI. The GUI manipulation specification concerns now the wd-components `Week`, `Day`, and `Talk`. We implement the same structure manipulation functionality as in the previous sections (i.e., swapping two adjacent components, moving a component to the beginning or the end of a container), this time for both talks and days—

[3] Note that when `b` and `c` are adjacent, they are necessarily in the same container, hence one could have defined the argument `talks` in terms of `c`.

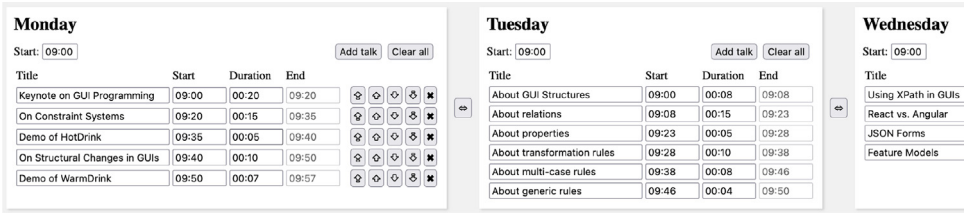

Fig. 5. A screenshot of the extended conference planning application, where both days and talks within a day can be swapped. The buttons for swapping talks appear for the day that is hovered by the mouse, e.g., Thursday in this screenshot.

```
structure
 root ConferenceApp {
  week: Week
 }
 Week [[ var start ]] {
  days: Day*
 }
 Day [[ ref prev, var day, var start, var duration ]] {
  talks: Talk*
 }
 Talk [[ var start, var duration, var end, ref prevStart,
        ref prevDuration ]]
```

Listing 9: The specification of the conference planner application structure.

```
component Week {
 var start = 0;
}
component Day {
 var number, name, &prev, start = 0;
 constraint {
  increment(prev -> number) => prev + 1;
 }
 constraint {
  updateDayName(number -> name) => {
   const days = ["Monday", "Tuesday", ...];
   return days[(number - 1) % 7];   }
 }
}
```

Listing 10: The HotDrink specification for the cs-components `Week` and `Day`.

```
relations
 precedesDay(Day a, Day b) {
  test { hasNext(a, b) }
  establish {
   insertAfter(b, a);
   b.prev = a.day;
  }
 }
 precedesTalk(Talk a, Talk b) {
  test { hasNext(a, b) }
  establish {
   insertAfter(b, a);
   b.prevStart = a.start;
   b.prevDuration = b.duration;
  }
 }
```

Listing 11: The definitions of relations `precedesDay` and `precedesTalk`.

```
swap<FirstIn, Precedes, IsLast>(
 any a = b/preceding-sibling, any b, any c,
 any d = c/following-sibling, any cont = b/container) {
 case a Precedes b, b Precedes c, c Precedes d =>
       a Precedes c, c Precedes b, b Precedes d
 case b FirstIn cont, b Precedes c, IsLast c =>
       c FirstIn cont, c Precedes b, IsLast b
 case b FirstIn cont, b Precedes c, c Precedes d =>
       c FirstIn cont, c Precedes b, b Precedes d
 case a Precedes b, b Precedes c, IsLast c =>
       a Precedes c, c Precedes b, IsLast b
}
```

Listing 12: The definition of the parameterized multi-case rule `swap`.

the same parameterized transformation rules suffice for manipulating both kinds of structural components.

The WarmDrink specification of the conference planner application structure, shown in Listing 9, is as follows. The wd-component `Talk` is the same as in the conference day planner introduced in Section 3.1. A corresponding constraint system component is associated with each instance of `Week`, `Day`, and `Talk`, respectively. The HotDrink specification for cs-component `Talk` is the same as the one in Section 2.2, `Week` and `Day` are defined in Listing 10. The `number` variable is the weekday as an ordinal, `name` its familiar name. The `prev` reference points to the previous day's `number` (except for the first day, for which it points to the week's `start` variable). The `increment` constraint thus advances the day from the previous, and `dayName` constraint guarantees that `name` has the correct weekday.

Since we now have two different substructures, we need different relations for days and talks that express what it means for one component to immediately precede another; we define `precedesDay` and `precedesTalk` in Listing 11. Note that `Day` and `Talk` cannot share the same (parameterized) precedes relation because of the different reference bindings in the establish block.

Now, instead of defining two multi-case rules, the one for swapping `Day` instances and the other `Talk` instances, we define one parameterized rule in Listing 12. Both multi-case and standard rules can be parameterized on relations. A parameterized rule is syntactically equal to a non-parameterized rule but additionally specify a list, enclosed in angle brackets, of relations on which it is parameterized on.

A parameterized rule is instantiated by specifying a concrete relation name for each of the relation parameters. The name of an

```
insert<FirstIn, Precedes, IsLast, NotInContainer>(
  any cont, any a, any b, any c) {
 case a Precedes c, b NotInContainer cont =>
       a Precedes b, b Precedes c
 case c FirstIn cont, b NotInContainer cont =>
       b FirstIn cont, b Precedes c
 case IsLast a, b NotInContainer cont =>
       a Precedes b, IsLast b
 case b NotInContainer cont =>
       b FirstIn cont
}
```

Listing 13: The definition of the parameterized multi-case rule `insert`.

instantiated rule, given after the `as` keyword, becomes the name of the generated JavaScript function. A rule parameterized on different relations may still have the same type signature in the generated JavaScript functions, and thus, all instantiations must have unique names.

The conference application instantiates the parameterized `swap` rule twice:

```
instantiate swap<isFirstTalkIn, precedesTalk,
    isLastTalk> as swapTalks
instantiate swap<isFirstDayIn, precedesDay, isLastDay>
    as swapDays
```

The relations `isFirstTalkIn` and `isLastTalk` are those defined in Section 4.1; the relations `isFirstDayIn` and `isLastDay` are defined in a similar way.

Our running example uses two more transformation rules, `insert` for inserting a new element in a container, `remove` for the opposite. We show the `insert` rule, `remove` is a similar generic multi-case rule. The `insert` rule, shown in Listing 13, is parameterized over four relations and defined in terms of the four arguments `cont`, `a`, `b`, and `c`. The rule inserts `b` between `a` and `c` in their container `cont`. It again distinguishes between different cases: `b` inserted (1) between two elements, (2) as the first element, (3) as the last element, and (4) as the only element. We instantiate `insert` for inserting talks and days as follows.

```
instantiate insert<isFirstDayIn, precedesDay, isLastDay,
     dayIsNotInContainer> as insertDay
instantiate insert<isFirstTalkIn, precedesTalk,
    isLastTalk, talkIsNotInContainer> as insertTalk
```

5. From WarmDrink specifications to JavaScript code

5.1. Code generated from a WarmDrink specification

The transformation rules specified in WarmDrink produce an API in the host language (JavaScript) for manipulating GUI structures[4]. Table 1 summarizes how wd-components, functions, predicates, relations, and transformation rules from a WarmDrink specification $\langle \mathbf{S}, \mathbf{F}^{\mathrm{js}}, \mathbf{R}, \mathbf{T} \rangle$ are transpiled into JavaScript.

For a *structure* specification **S**, a JavaScript object `root` is generated. Its keys are the names of **S**'s features: nested components become objects and features with cardinality $*$ become arrays.

[4] The WarmDrink library's source code can be found on the git-repository https://git.app.uib.no/warmdrink-cola/warmdrink-ide.

Table 1
An overview of how concepts in a WarmDrink specification are transpiled into JavaScript. The generated exported functions are used by an application programmer to implement structure manipulations.

WarmDrink specification $\langle \mathbf{S}, \mathbf{F}^{\mathrm{js}}, \mathbf{R}, \mathbf{T} \rangle$	Generated JavaScript	Exported?
Structure **S**	Object `root` whose keys are names of the features in **S**	
wd-component C_i in **S**	Function `new`C_i`(view)` that creates a semantic GUI component and establishes references between it, the corresponding cs-component, and the DOM element `view`	✔
Function/predicate `f(`a_1`, ..., `a_n`)` in $\mathbf{F}^{\mathrm{js}}$	Function `WD_FUNC__f(`a_1`, ..., `a_n`)` whose body contains the JavaScript code specified in `structure`- and `view`-blocks of `f`, with spliced x-expressions expanded	
Relation `r(`C_1 c_1`, ..., `C_n c_n`)` in **R**	Object `WD_RELATION__r` with keys `test`, `establish`, `unestablish`, each of which is an anonymous function	
Transformation rule `t(`C_1 c_1`, ..., `C_n c_n`)` in **T**	Function `t(`c_1`, ..., `c_n`)` checking rule's premises, then unestablishing all of them, and then establishing all consequences and testing that they hold	✔
Multi-case rule `t(`C_1 c_1`, ..., `C_n c_n`)` in **T**	Function `t(`c_1`, ..., `c_n`)` whose body has a conditional statement for each case, checking whether the premises of that case hold	✔
Parameterized rule `t<`r_1`, ..., `r_n`>(any `c_1`, ..., any `c_n`)` in **T**	Function `t(`r_1`, ..., `r_n`)` that returns an anonymous function with signature `(`c_1`, ..., `c_n`)` whose behavior is analogous to functions generated for ordinary rules	
Rule instantiation `t<`rel_1`, ..., `rel_k`>` as t_{inst} of a parameterized rule `t<`r_1`, ..., `r_k`>(any `c_1`, ..., any `c_ℓ`)` in **T**	Function t_{inst}`(`c_1`, ..., `c_ℓ`)` invoking `t(`rel_1`, ..., `rel_n`)(`c_1`, ..., `c_ℓ`)`	✔
Rule instantiation of a parameterized multi-case rule	Same as previous	✔
WarmDrink language constructs within specifications of relations		
x-expression $x_1/x_2/\cdots/x_\ell$	Expression $x_1.x_2.\cdots.x_\ell$	*n/a*
Function call `f(`c_1`, ..., `c_n`)`	Function call `WD_FUNC__f(`c_1`, ..., `c_n`)`	*n/a*
Constraint system reference update $a.x = b.y$;	a`._model.vs.`x `=` b`._model.vs.`y`;` a`._model.system.update();`	*n/a*

For each *wd-component* C in a structure, an exported function `new`C`(v)` is generated. This function creates an instance $\widehat{C}$ ready to be inserted into the structure, and sets references from $\widehat{C}$ to the corresponding view `v` and constraint system component $\mathsf{Model}(\widehat{C})$, and from `v` to $\widehat{C}$.

Each *function* `f` declared in WarmDrink is transpiled into a JavaScript function `WD_FUNC__f` with the same signature. This generated function contains the JavaScript code from the `structure`- and `view`-blocks of `f`. Spliced x-expressions of the form $x_1/x_2/\cdots/x_\ell$ that appear in these blocks are *expanded* into JavaScript code of the form $x_1.x_2.\cdots.x_\ell$, where x_i is either a feature which is expanded[5]

[5] Note that all spliced x-expressions that appear within a function are precomputed and stored in variables. This is because the `structure`-, `view`-blocks of a function must start their execution in the same state of the populated structure $\widehat{S}$. Otherwise, changes to $\widehat{S}$ in the `structure`-block

```
function WD_FUNC__insertAfter(b, a) {
 const __a__container = a?.container;
 const __a__following_sibling = a?.following_sibling;
 // structure
 const indexA = __a__container.indexOf(a);
 const indexB = __a__container.indexOf(b);
 if (indexB !== -1) __a__container.splice(indexB, 1);
 __a__container.splice(indexA + 1, 0, b);
 // view
 __a__container?._view.insertBefore(b?._view,
    __a__following_sibling?._view);
}
```

Listing 14: The JavaScript function generated from the WarmDrink function `insertAfter`.

into a JavaScript object key, or one of the axes defined in Section 4.1. Wd-components have a getter function for each axis; the getters are attached at the components' initialization. E.g., `a/container/first-child` in WarmDrink is expanded to `a?.container?. first_child` (the JavaScript operator "`?.`" short-circuits to null if the left operand is null). Listing 14 shows the JavaScript function generated from the WarmDrink function `insertAfter` defined in Listing 4.

For each k-ary *relation* r, a JavaScript object `WD_RELATION__`r with three keys, `test`, `establish`, and `unestablish`, is generated. Each of these keys store an anonymous function with k arguments. The first function is a predicate that checks whether each statement in the `test`-block of the relation specification returns a true value. The second and the third functions are symmetrical: they establish (unestablish) the relation r, by executing the imperative code specified in the `establish` (`unestablish`) block of the specification of r.

Within a relation specification, *call to a WarmDrink function* f is transpiled to a JavaScript function call `WD_FUNC__`f. The *assignments* to cs-component references of the form a`.x = `b`.y` are transpiled into JavaScript as follows:

a`._model.vs.x = `b`._model.vs.y;`

a`._model.system.update();`

The `vs` member of the `_model` gives access to a cs-component's variables and its `system` member to the underlying constraint system. The `system.update()` call forces HotDrink to enforce all constraints. Listing 15 shows the JavaScript code generated for the relation `precedesTalk`.

Each transformation rule t is transpiled into an exported JavaScript function as follows. For an ordinary rule defined on arguments $a_1, \ldots, a_k$, the signature of the generated function is $t(a_1, \ldots, a_k)$. This function first checks the rule's premises by calling the corresponding `test` member for every premise. If the premises hold, it invokes `unestablish` for every premise and `establish` for every consequence of the rule. After that, the function invokes `test` for every consequence, checking that they have indeed been established. An error is reported if any of the checks fails.

For a multi-case rule with arguments $a_1, \ldots, a_k$, and implicit arguments $w_1, \ldots, w_m$ initialized with x-expressions $x_1, \ldots, x_m$, the signature of the generated exported function is $t(a_1, \ldots, a_k)$. First, the parameters w_i are initialized with values computed by evaluating x_i. For each case of the rule, an `if-then` statement is generated that checks whether the premises of that case hold. The body of the conditional statement is analogous to the body of the function

might rearrange the structural elements and lead to incorrect behavior in the two other blocks. With the spliced x-expressions precomputed, the order of the two blocks is not significant.

```
const WD_RELATION__precedesTalk = {
 test: (a, b) => (true && WD_FUNC__hasNext(a, b)),
 establish: (a, b) => {
  WD_FUNC__insertAfter(b, a);
  b._model.vs.prevStart = a._model.vs.start;
  b._model.vs.prevDuration = a._model.vs.duration;
  b._model.system.update();
 },
 unestablish: (a, b) => { }
}
```

Listing 15: The JavaScript function generated for the WarmDrink relation `precedesTalk`. The establish block invokes the JavaScript function generated from `insertAfter`, updates a reference binding between the two corresponding cs-components of `a` and `b`, and notifies the constraint system that there was a change.

```
function insert(FirstIn, Precedes, IsLast,
    NotInContainer) {
 return (cont, a, b, c) => {
  if (true && c && a && cont && b && Precedes.test(a, c) &&
      NotInContainer.test(b, cont)) {
   Precedes.unestablish(a, c);
   NotInContainer.unestablish(b, cont);
   Precedes.establish(a, b);
   Precedes.establish(b, c);
   if ( !(Precedes.test(a, b) && Precedes.test(b, c)) ) {
    throw error("Failed to apply rule ...");
   }
  }
  else if (true && c && cont && b && FirstIn.test(c, cont)
    && NotInContainer.test(b, cont) ) { ... }
  else if (true && a && cont && b && IsLast.test(a) &&
    NotInContainer.test(b, cont) ) { ... }
  else if (true && cont && b && NotInContainer.test(b,
    cont) ) { ... }
  else { console.error(
    "Failed: couldn't match any cases ...") }
 }
}
```

Listing 16: JavaScript code generated for the parameterized multi-case rule `insert`. The anonymous function inserts element `b` between elements `a` and `c` in container `cont`.

generated from an ordinary rule: it unestablishes the premises and establishes the consequences. The conditional statement is concluded with a return statement: this guarantees no fall-through the other cases of the multi-case rule.

For a rule parameterized on abstract relations $R_1, \ldots, R_m$ and defined on arguments $a_1, \ldots, a_k$, the generated JavaScript function has signature $t(R_1, \ldots, R_m)$. It returns an anonymous function with signature $(a_1, \ldots, a_k)$, whose behavior is analogous to functions generated for non-paramaterized rules. Listing 16 shows an example of the JavaScript function generated from the parameterized multi-case rule `insert` shown in Listing 13.

For a rule t_{inst} that instantiates a parameterized rule $t\langle R_1, \ldots, R_k\rangle(a_1 \cdots a_\ell)$ with relations $\text{rel}_1, \ldots, \text{rel}_k$, the generated exported JavaScript function has signature $t_{\text{inst}}(a_1, \ldots, a_\ell)$. Its body is of the form `return` $t(\text{rel}_1, \ldots, \text{rel}_k)(a_1, \ldots, a_\ell)$, i.e., it calls the anonymous function returned by $t(\text{rel}_1, \ldots, \text{rel}_k)$ with the arguments $a_1, \ldots, a_\ell$.

The generated function for an instantiation of a parameterized multi-case rule is exactly the same as for an instantiation of a parameterized ordinary rule.

Finally, two auxiliary JavaScript functions are generated: `anchor`, that locates the structural component corresponding to a current view, and `get`, that identifies components in relation to the anchor, using x-expressions for navigating component trees. The machinery of these functions is explained in detail in the next subsection.

5.2. Running example: the generated JavaScript API

Of importance to the application programmer are the exported JavaScript functions which are generated from WarmDrink transformation rules. Each such generated function has the same name as the corresponding transformation rule and the same number of arguments, which are the wd-components on which the transformation rule is to be applied.

Consider the code below implementing an event handler `appendNewTalk` that reacts to clicks on the buttons for adding a new talk at the end of a day in the running example (see Fig. 3).

```
1  function appendNewTalk(event) {
2   const talkView = (<li> // create view nodes using JSX
        <input data-bind="title" />
3
4     <input data-bind="start" />
5     <input data-bind="duration" />
6     <input data-bind="end" disabled />
7    </li>);
8   const t = newTalk(talkView);
9   anchor(event.target);
10  insertTalk(get("./talks"), get("./talks/last-child"),
      t, null); }
```

In order to append a new talk, one must first create its view (lines 2–7), then bind the view to the corresponding HotDrink and WarmDrink components (function `newTalk` in line 8), and finally invoke the WarmDrink transformation rule `insertTalk` (lines 9–10).

Function `newTalk` in line 8 constructs an instance of the wd-component `Talk`. It expects a view (DOM-element) to be passed as the argument, creates a wd-component and an instance of the corresponding cs-component, binds the latter's variables to DOM elements in `talkView`, and establishes references between the wd-component and cs-component, and form the `talkView` to the wd-component.

For navigating and accessing components in the populated structure (using x-expressions), the generated JavaScript API has functions `anchor` and `get`. An invocation of `anchor` locates the wd-component corresponding to the current view, and sets it as a reference point for navigating the populated structure.[6] Line 9 invokes `anchor` with the clicked button to set the currently operated on wd-component `Day` as the reference point. Subsequent calls to `get` identify components in relation to the reference point. In line 10, `get("./talks")` returns the current day's container of talks and `get("./talks/last-child")` the last element of this container. With these arguments as context, the new talk can be inserted after the last element. The `insertTalk` function takes care of all bookkeeping and changes in the cs-component and the view. This function has the same arguments as the `insertTalk` (Section 4.4) transformation rule from which it was generated: a list of talks, the talk that should precede the inserted talk, the talk to be inserted, and the talk that should follow the inserted talk. Here the last argument is `null`, as the talk is inserted at the end.

As examples of using two different instantiations of a parameterized rule we show code that swaps adjacent talks and code that swaps agendas of adjacent days. The first example first sets the anchor and then performs two swaps: the first swaps a talk with the talk that follows it, the second swaps them back.

```
anchor(t);
 // assume t is in the view of the talk of interest
swapTalks(get("."), get("./following-sibling"));
 // move talk down
swapTalks(get("./preceding-sibling"), get("."));
 // move talk up
```

Code for swapping the agenda of two adjacent days of the week – that is, shifting a day to the left or to the right – differs only on the chosen name of the rule instance:

```
anchor(d);
 // assume d is in the view of the day of interest
swapDays(get("."), get("./following-sibling"));
 // shift day right
swapDays(get("./preceding-sibling"), get("."));
 // shift day left
```

5.3. Architecture of a WarmDrink-based application

Fig. 6 shows a sketch of an application developed with WarmDrink. It comprises a HotDrink specification (see Fig. 6*a*) and an API generated from a WarmDrink specification (Fig. 6*b*) that are used in an application's main file (Fig. 6*d*) and an HTML file with the application's view (Fig. 6*c*).

The HotDrink specification file exports definitions of the cs-components, which are then imported by WarmDrink-generated API. For each wd-component that the WarmDrink program specifies to have a cs-component, the HotDrink file must declare a HotDrink component with the same name and must include the variables and variable references specified in the WarmDrink program.

The API is used by the application programmer to implement functionality that deals with manipulating structures in the GUI of the application. This is done in the application's main file, where GUI event listeners are defined. The HTML file loads the application's JavaScript file and defines the skeleton of the GUI view. The body of the HTML file includes a `div`-tag with the attribute `data-warmdrink-root` to indicate the populated structure's root element.

5.4. The WarmDrink IDE

We have implemented an integrated development environment (IDE) for writing WarmDrink code. The IDE handles transpiling WarmDrink programs to JavaScript, and provides common IDE-features for the language, such as validation, syntax highlighting, and suggestions.

The WarmDrink IDE is implemented with the language workbench Eclipse Xtext [27]. Based on specifications of a language's syntax and typing rules, a language workbench [28] produces a code generator and tailored IDE with standard services, including a syntax-aware editor, code completion, code folding, automatic code corrections, and basic code refactoring [29]. From the grammar specification for WarmDrink, Xtext generates a model using Eclipse Modeling Framework [26]. The model is populated during parsing, producing an abstract syntax tree that can be further analyzed or transformed. We use Eclipse Xtend's [30] transformation language to generate the JavaScript code. Fig. 7 presents a screenshot of a working Eclipse instance of WarmDrink.

[6] The links from the view to a wd-component make it possible to access the component with ease, e.g., from the target object of an event handler. For example, the event handlers of the `button`-elements in Fig. 5 find the matching structural elements through these references: starting from the clicked button element, the DOM-tree is walked up to find an element that has a reference to a wd-component instance.

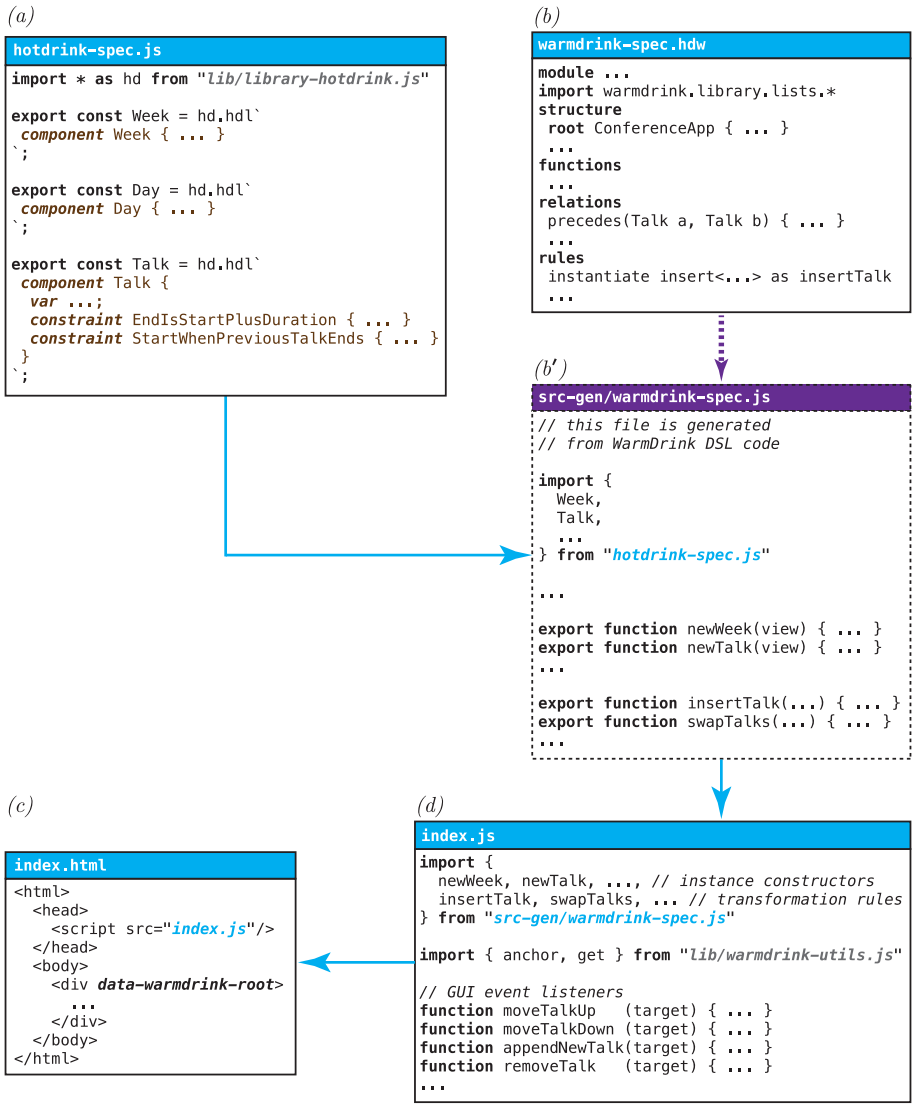

Fig. 6. Architecture of an application developed with HotDrink and WarmDrink: *(a)* specification of HotDrink components; *(b)* WarmDrink specification; (*b*′) API generated from the WarmDrink specification; *(c)* application's HTML file; *(d)* application's JavaScript file. Solid arrows designate imports. Dashed arrow designates code generation. The application programmer writes parts *a*, *b*, *c*, and *d*.

The IDE for WarmDrink performs validation of the source code and reports standard code issues, such as duplicate declarations or undeclared identifiers. In addition, we have implemented a wide range of WarmDrink-specific validations, such as checking that there are no cycles in a structure declaration; checking that types of components match; type-checking the return types of functions in `test`- and `(un)establish`-blocks of relations; correct scoping in constraint system assignments (for example, a constraint system variable in the left hand side must be a reference, and only eligible references are shown in autocomplete menus); inferring types and typechecking arguments in parameterized rules for each of their instantiations; and so on. In addition, the IDE supports automatic code corrections – "quickfixes" – that can be used both to correct an erroneous code fragment (e.g., by suggesting identifiers in a correct scope whenever an undeclared identifier is met).

Code written in WarmDrink can be stored in modules, and modules can import other modules. Importing makes wd-components, functions, relations, properties, and rules visible to the importing module. Modules enable creating a standard library of transformation rules for common structural manipulations (such as swapping and inserting elements in lists).

6. Evaluating WarmDrink

The GUI programming community seems to always be chasing the ultimate GUI architecture; there is a long history of different patterns with different trade-offs. To evaluate the experience of programming GUIs with WarmDrink and compare it with alternative approaches, we implemented the conference planner application in three different ways: (*i*) as a React application, without the use of any constraint system library; (*ii*) as a React application that uses the HotDrink library to manage the dataflow between widgets in the GUI; and (*iii*) as a JavaScript application that uses HotDrink to manage the dataflow and WarmDrink to manage the structural changes in the GUI.

There are obviously several more implementation choices, including using plain JavaScript without using any libraries or frameworks, using HotDrink without WarmDrink, or using any of the many popular GUI frameworks [10,31–33]. Approaches that do not use any modern GUI framework leave many low-level details as the application programmer's responsibility, which is the reason for not considering a plain JavaScript implementation. Amongst the popular GUI frameworks, we chose React as a point of comparison, as many consider it to be a representative of the state of the art today.

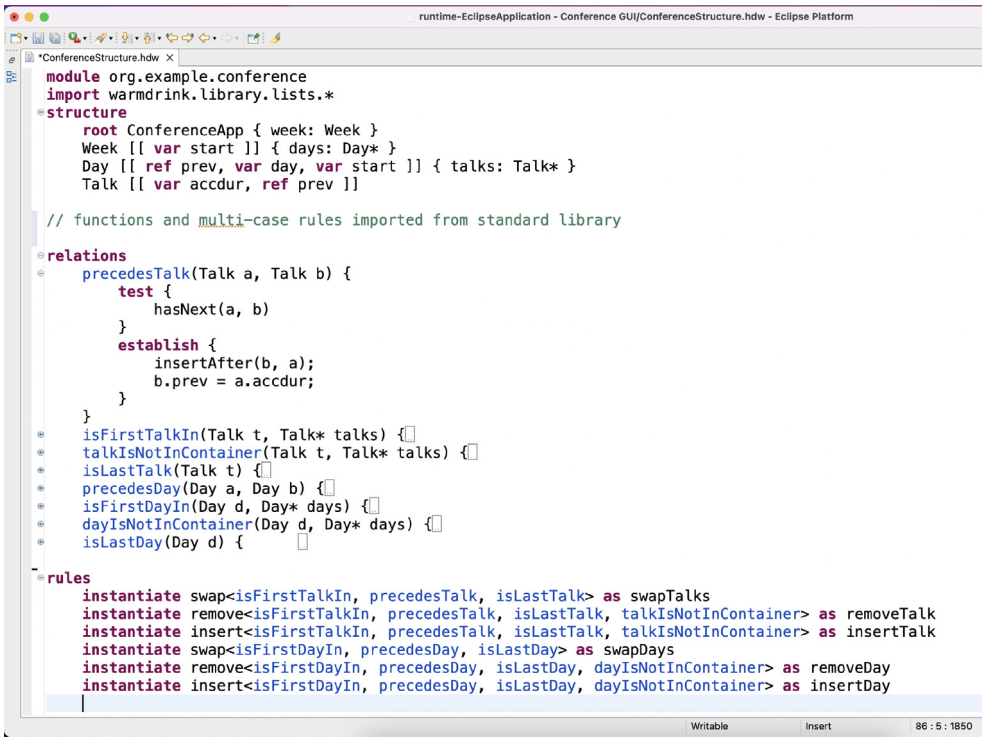

Fig. 7. A screenshot of the Eclipse IDE for WarmDrink. The IDE, implemented with Eclipse Xtext, supports syntax highlighting, auto-completion, hyperlinking, hover tooltips, and rename refactoring.

To evaluate different GUI implementation approaches and how resilient they are to minor changes in requirements, we implement the same conference planner application in the three approaches (i)–(*iii*) and subject the implementations to two requirement changes R_1 and R_2. In R_1, we introduce an *author* to every talk and, whenever two consecutive talks share the same author, give a warning to the user. R_2 provides more flexibility for scheduling talks: while the base conference planning application allows the user to modify only the *duration* of each talk, from which the *ending time* (and the subsequent *start times*) are computed, the new GUI lets the user edit the *start time* of any talk—and the GUI is expected to adjust the start times of the earlier and later talks.

With this experiment, we seek to answer the question whether our approach with *explicit specifications of structural changes in "dataflow-rich" GUIs leads to succinct and intuitive GUI implementations, where minor changes in requirements mean minor changes in implementation.*

Base implementations

The implementation approach (*i*) uses only React and plain JavaScript; the programmer gets no support for maintaining relations between different state properties from a constraint system. In the conference planner application this means that the code that, say, inserts a new talk loops over all talks after the insertion point and modifies their start times.

In **the implementation approach (*ii*),** we store the HotDrink's cs-components in the states of React components whose structure matches the constraint system. Subscriptions to cs-components' variables trigger state changes in the corresponding React components, so that React render functions keep the view in sync with the constraint system.

Modifying the structure of the components (adding or reordering conference days or talks) is left to the GUI programmer. For example, the `moveUp` function in Listing 17 that swaps a talk at index `ind` with the preceding talk, accesses the list of HotDrink cs-components in the React component's state, creates a new copy of that list, ensures that the talk to be moved up is *not* the first talk, updates the connections between cs-components, swaps the two elements in the new list, and updates the React component's state with that list, to be rendered by React. Other structure-manipulation operations require similar code.

In **the implementation approach (*iii*),** the functionality for swapping, inserting, and removing talks is implemented by using the WarmDrink relations and instantiating the parameterized rules `swap`, `insert`, and `remove`.

Implementing requirement change R_1

In approach (*i*), a helper function constructs a new list of talks calculating their start and end times in one loop, and is used by all operations that perform structural changes or handle user edits. To implement R_1, this loop has to check whether the current talk's author is the same as the previous talk's author, and, if so, present a warning to the user.

To implement R_1 in approach (*ii*), we add an extra constraint in the HotDrink specification that indicates when the no-same-author requirement between consecutive talks is violated. The constraint has references to the current and previous talks' authors; the reference to the previous talk's author needs to be kept consistent after every structural change. Therefore, the implementations of all functions performing structural operations change.

To implement R_1 in approach (*iii*), the same HotDrink specification change is needed. The relations `precedesTalk`, `isFirstTalkIn`, and `talkIsNotInContainer` are modified to describe the changed linking between two consecutive talks. The transformation rules remain unaltered.

```
moveUp(ind) {
  const tmpTalkList = [...this.state.hdTalks];
  if (ind === 0) {
    throw Error("Cannot move first talk up...");
  }
  tmpTalkList[ind-1].vs.prevStart =
    tmpTalkList[ind].vs.start;
  tmpTalkList[ind-1].vs.prevDuration =
    tmpTalkList[ind].vs.duration;
  if (ind === 1) {
    tmpTalkList[ind].vs.prevStart =
      this.state.hdStart.vs.start;
    tmpTalkList[ind].vs.prevDuration =
      this.state.hdStart.vs.duration;
  } else {
    tmpTalkList[ind].vs.prevStart =
      tmpTalkList[ind-2].vs.start;
    tmpTalkList[ind].vs.prevDuration =
      tmpTalkList[ind-2].vs.duration;
  }
  if (ind < tmpTalkList.length-1) {
    tmpTalkList[ind+1].vs.prevStart =
      tmpTalkList[ind-1].vs.start;
    tmpTalkList[ind+1].vs.prevDuration =
      tmpTalkList[ind-1].vs.duration;
  }
  defaultConstraintSystem.update();
  const moved = tmpTalkList.splice(ind, 1)[0];
  tmpTalkList.splice(ind-1, 0, moved);
  this.setState({ hdTalks: tmpTalkList });
}
```

Listing 17: The JavaScript function `moveUp` from the implementation approach (*ii*). When the function is called with the index of a talk, the talk is moved one step up in the list of talks.

Implementing requirement change R_2

After R_2, editing start time and editing duration lead to different dataflows. In approach (*i*), this implies more complicated looping structures. We add a new helper function to deal with changes to starting times, which computes start times towards the beginning and towards the end of the day in two separate loops.

In approaches (*ii*) and (*iii*), a small change to the HotDrink specification's `AdjacentTalks` constraint defined in Listing 1 suffices. In the new definition below, we add a new method (line 4) to compute the previous talk's start time from its duration and the current talk's start time.[7]

```
1 constraint AdjacentTalks {
2   (prevStart, prevDuration -> start) =>
3     addTimes(prevStart, prevDuration);
4   (start, prevDuration -> prevStart, prevDuration) =>
        [subtractTimes(start, prevDuration),
       prevDuration];
5 }
```

[7] The second method includes `prevDuration` as output to ensure that the constraint system chooses the dataflow defined by the first method when the user edits the talk duration, and the one defined by the second when the user edits the start time.

Discussion

The main difference between the approach (*i*) and the other two approaches is that in the latter, the dependencies between variables in different talk components have a concrete programmatically accessible representation as the constraint system, while in the former, they do not. In other words, without a constraint system, the programmer writes an algorithm, in any way convenient, that computes new values for the variables in all talks after a user edit or a structural change, whereas with a constraint system, the programmer specifies a structure of dependencies that keep variables' values consistent, and how to modify that structure (concisely expressing these modifications is WarmDrink's raison d'être). The dependency and structure specifications make the implementations with approaches (*ii*) and (*iii*) slightly longer[8] than that with (*i*), but on the other hand, code modifications due to new requirements that affect the dataflow are *localized* in these specifications, instead of requiring a new algorithm for value updates.

In approach (*i*), the change of the update algorithm due to R_1 is minor, but R_2 requires a completely new algorithm. In approach (*ii*), since every structural operation is responsible for maintaining the dependency representation, implementing R_1 requires a large number of changes— each of these operations must be modified. In approach (*iii*), these changes are not needed because structural transformations are defined in terms of a small number of relations; only the definitions of these relations are affected. Implementing R_2 is a one-line change in both approaches (*ii*) and (*iii*).

We remark that approach (*iii*) satisfies the goal of minor changes in the requirements leading to minor changes in the implementation: neither R_1 nor R_2 affect the high-level structure of the data that the GUI displays and indeed no changes are needed in code that deals with structural changes.

Our experiment, which is somewhat idealized to emphasize implementing and maintaining the dataflow aspect of a GUI, gives us some assurance of the benefits of explicitly specifying GUI structures and structural changes, but, of course, not conclusive evidence. To dispel threats to validity of our evaluation, we eventually need to gain experience with WarmDrink-based implementations of "industrial strength" GUIs with all their varied problems and nuances.

7. Related work

Throughout the text we relate WarmDrink with contemporary GUI programming frameworks, patterns, and approaches. We also describe how our work builds on language work benches and other DSL technologies. In this section we describe a few more connections to prior research.

Many programming approaches aspire to make application programming primarily to be about assembling together predefined components, describing their connections declaratively. The fairly recent Déjà Vu [34] framework is a concrete realization of this idea: fully componentized micro-services implement high-level *concepts* that programmers instantiate and plug together to applications. Like most component approaches, this approach is silent about manipulating the structures composed of connected concepts. This is not surprising: Déjà Vu's concepts encapsulate substantially complex functionalities (geolocation, scheduling times, authentication) that would perhaps not often appear as repeating parts of another complex structure.

Several other approaches pursue the goal of clear and unmuddled specifications of GUI structures, but choose a different kind of generative approach from ours. For example, the JavaScript library JSON Forms [35] and the WebDSL [36] language start from concise high-level specifications of structure (and some behavior), and generate full GUI implementations based on these specifications. The generated GUI structures can come with a rich set of features, such as entry validation,

[8] About 10 lines of HotDrink code and 40 lines of WarmDrink code.

```
Module      ::= module ID
                ( import ID↪Module )*
                structure Component+
                functions Function*
                relations Relation*
                rules Rule*
Component   ::= [root] ID [CSSpec] { Feature* }
CSSpec      ::= [[ (ref ID | var ID)+ ]]
Feature     ::= ID : ID↪Component [*]
Function    ::= (bool | void) ID ( (any ID )* ) {
                    [structure { JSCode }] [view { JSCode }] }
Relation    ::= ID ( Arg+ ) { test Code
                    establish Code unestablish Code }
Arg         ::= (ID↪Component [*] | any) ID
Code        ::= FunctionCall |
                CSRefUpd
FunctionCall ::= ID↪Function ( ID*↪Arg )
CSRefUpd    ::= ID↪Arg . ID↪Ref = (ID↪Arg | | XExpr |) .
                    (ID↪Ref | ID↪Var)
Rule        ::= Signature { Body } |
                Instantiation
Signature   ::= ID ( Arg+ ) |
                ID <ID+> ( (any ID [= XExpr])+ )
Body        ::= Predicate* => Predicate* |
                ( case Predicate* => Predicate* )+
Instantiation ::= instantiate ID↪Rule < ID+↪Relation > as ID
Predicate   ::= [ID↪Arg] ID↪Relation ID+↪Arg
XExpr       ::= (root | ID↪Arg) (/ (ID↪Feature |
                parent | container |
                preceding-sibling | following-sibling |
                first-child | last-child ))*
JSCode      ::= js ’’’ ··· interpolated string ··· ’’’
```

Fig. A.8. An outline of the concrete syntax grammar of the WarmDrink language. Optional sequences of terminals and nonterminals are enclosed within square brackets. Notation $ID_{\hookrightarrow A}$ designates an identifier that is a cross-reference to an identifier declared in a substring derived from nonterminal A.

layout, access control, and structure modification behaviors, but the generator determines the implementation of the structures and along with it much of the application too. WarmDrink, instead, can impose an explicit structure representation over implicit GUI structures, and generate code for manipulating those structures.

Regarding constraint systems, we use HotDrink [14] to maintain the underlying data dependencies in the examples of this paper. There are other constraint system-based libraries and languages that could be used in a similar fashion, but would require modifying the WarmDrink DSL and its JavaScript generator. For example, ConstraintJS [37] is another JavaScript dataflow constraint library, and Babelsberg a general framework for integrating constraint systems into object-oriented languages [38]. HotDrink's handling of multi-way dataflow constraint systems is based on the QuickPlan solver algorithm [17].

Our approach arises from practical programming concerns: we impose a (tree) structure over components that appear in a GUI's models and views, and provide structural operations over it. Although our implementation uses trees to structure user interfaces, programmers may want to define more complicated aggregation relationships between graphical components than containment. We have tentatively investigated hypergraphs for a general basis of populated structures, with vertices representing GUI components and edges n-ary relations between the components, and hyperedge replacement grammars [39] for specifying modifications in such structures. Such grammars may be a well-fitting formalism for expressing a larger set of structures, and with more precision (e.g., a list with different component types alternating is easily expressible).

Using a hypergraph as the underlying populated structure would necessitate a more complex mapping from the structure's vertices to the visual UI elements. Such mappings could be specified with data dependency algebras (DDA) [40–42]. With DDAs, programmers could define, for instance, grid structures where hyperedges relate cells to their cardinal neighbors. Further investigation of these connections remains as future work.

8. Conclusion

This paper is an exploration of a new approach for managing the complexity of GUI programming. Much of this complexity can be attributed to tasks that manipulate various structures that appear on GUIs, in particular because the same structures have multiple projections to models and views. Our work is a step towards understanding how structures in GUIs manifest as different projections to views and models. By providing a DSL that lets the programmer manipulate these different projections as one structure, GUI programming is simplified.

The DSL presented in this paper allows the programmer to make explicit GUI structures that would otherwise be incidental and implicit. With the DSL the programmer specifies a (semantic) structure, defines relations between elements in the structure, and defines rules for manipulating the structure in terms of how these relations change.

The relation specifications capture the intricacies of how relations are established and unestablished in views and models, in particular how links, references, and other connections between the elements change with these operations. The actual transformation rules remain clear and concise, and it is therefore easy to provide an extensive set of operations, a comprehensive API, for making changes to the structure.

For the application programmer, the API presents a semantic structure that can be navigated with x-expressions and manipulated with ease using high-level operations.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix. Outline of the WarmDrink grammar

See Fig. A.8.

References

[1] ApplyTexas web application https://www.applytexas.org (Accessed: 2022-11-24).

[2] B.A. Myers, M.B. Rosson, Survey on user interface programming, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '92, Association for Computing Machinery, New York, NY, USA, 1992, pp. 195–202, http://dx.doi.org/10.1145/142750.142789.

[3] S. Parent, A possible future for software development, in: Keynote Talk at the Workshop of Library-Centric Software Design, 2006, URL https://stlab.cc/legacy/figures/Possible_future.pdf.

[4] K.A. Stokke, M. Barash, J. Järvi, Manipulating GUI structures declaratively, in: Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, in: GPCE 2020, Association for Computing Machinery, New York, NY, USA, 2020, pp. 63–69, http://dx.doi.org/10.1145/3425898.3426956.

[5] B.A. Myers, Separating application code from toolkits: Eliminating the spaghetti of call-backs, in: Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology, 1991, pp. 211–220.

[6] G.E. Krasner, S.T. Pope, A cookbook for using the model-view controller user interface paradigm in smalltalk-80, J. Object Oriented Program. 1 (3) (1988) 26–49.

[7] J. Gossman, Introduction to Model/View/ViewModel pattern for building WPF apps, 2005, URL https://docs.microsoft.com/en-us/archive/blogs/johngossman/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps.

[8] M. Potel, MVP: Model-view-presenter: The Taligent programming model for C++ and Java, 1996, URL http://www.wildcrest.com/Potel/Portfolio/mvp.pdf.

[9] E. Czaplicki, The Elm Architecture, URL https://guide.elm-lang.org/architecture.

[10] J. Farrar, KnockoutJS Web Development, Packt Publishing, 2015.

[11] React—A JavaScript library for building user interfaces, 2020, https://reactjs.org/ (Accessed: 2020-05-02).

[12] M. Madsen, O. Lhoták, F. Tip, A semantics for the essence of React, in: R. Hirschfeld, T. Pape (Eds.), 34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference), in: LIPIcs, 166, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 12:1–12:26, http://dx.doi.org/10.4230/LIPIcs.ECOOP.2020.12.

[13] ReactiveUI—An advanced, composable, functional reactive model-view-viewmodel framework for all .NET platforms, https://www.reactiveui.net.

[14] J. Freeman, J. Järvi, G. Foust, HotDrink: A library for web user interfaces, SIGPLAN Not. 48 (3) (2012) 80–83, http://dx.doi.org/10.1145/2480361.2371413.

[15] G. Foust, J. Järvi, S. Parent, Generating reactive programs for graphical user interfaces from multi-way dataflow constraint systems, in: Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, in: GPCE 2015, ACM, New York, NY, USA, 2015, pp. 121–130, http://dx.doi.org/10.1145/2814204.2814207.

[16] J. Järvi, HotDrink, 2021, URL https://www.npmjs.com/package/hotdrink.

[17] B. Vander Zanden, An incremental algorithm for satisfying hierarchies of multi-way dataflow constraints, ACM Trans. Program. Lang. Syst. 18 (1) (1996) 30–72, http://dx.doi.org/10.1145/225540.225543.

[18] I.E. Sutherland, Sketchpad: A man-machine graphical communication system, in: DAC '64: Proceedings of the SHARE Design Automation Workshop, ACM, New York, NY, USA, 1964, pp. 6329–6346.

[19] B.A. Myers, R.G. McDaniel, R.C. Miller, A.S. Ferrency, A. Faulring, B.D. Kyle, A. Mickish, A. Klimovitski, P. Doane, The Amulet environment: New models for effective user interface software development, Softw. Eng. 23 (6) (1997) 347–365.

[20] B. Myers, D. Giuse, R. Dannenberg, B. Zanden, D. Kosbie, E. Pervin, A. Mickish, P. Marchal, Garnet: Comprehensive support for graphical, highly interactive user interfaces, Computer 23 (11) (1990) 71–85.

[21] M. Sannella, Skyblue: A multi-way local propagation constraint solver for user interface construction, in: UIST '94: Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology, ACM, New York, NY, USA, 1994, pp. 137–146.

[22] Apple Inc., View layout, 2022, https://developer.apple.com/documentation/uikit/view_layout (Accessed: 2022-02-02).

[23] J. Järvi, M. Marcus, S. Parent, J. Freeman, J.N. Smith, Algorithms for user interfaces, in: GPCE'09: Int. Conf. on Generative Programming and Component Engineering, ACM, New York, NY, USA, 2009, pp. 147–156, http://dx.doi.org/10.1145/1621607.1621630.

[24] J. Freeman, J. Järvi, W. Kim, M. Marcus, S. Parent, Helping programmers help users, in: GPCE'11: Int. Conference on Generative Programming and Component Engineering, ACM, New York, NY, USA, 2011, pp. 177–184, http://dx.doi.org/10.1145/2047862.2047892.

[25] J. Clark, S. DeRose, XML path language (XPath), W3C Recommendation, 1999, https://www.w3.org/TR/1999/REC-xpath-19991116/ (Accessed: 2021-04-19).

[26] D. Steinberg, F. Budinski, M. Paternostro, E. Merks, EMF : Eclipse Modeling Framework, Addison-Wesley, Upper Saddle River, N.J, 2008.

[27] M. Eysholdt, J. Rupprecht, Migrating a large modeling environment from XML/UML to Xtext/GMF, in: W.R. Cook, S. Clarke, M.C. Rinard (Eds.), Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA, ACM, 2010, pp. 97–104, http://dx.doi.org/10.1145/1869542.1869559.

[28] S. Erdweg, T. van der Storm, M. Völter, L. Tratt, R. Bosman, W. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, J. Woning, Evaluating and comparing language workbenches: Existing results and benchmarks for the future, Comput. Lang. Syst. Struct. 44 (2015) 24–47, http://dx.doi.org/10.1016/j.cl.2015.08.007.

[29] L. Bettini, Implementing Domain-Specific Languages with Xtext and Xtend, Packt Publishing, 2016.

[30] S. Efftinge, S. Zarnekow, Xtend—modernized Java, 2020, https://www.eclipse.org/xtend/ (Accessed: 2020-07-15).

[31] Elm programming language, 2021, URL https://elm-lang.org (Accessed: 2021-04-17).

[32] Google LLC, Angular, 2020, https://angular.io/ (Accessed: 2020-07-15).

[33] Svelte: Cybernetically enhanced web apps, 2022, https://github.com/sveltejs/svelte/ (Accessed: 2022-06-01).

[34] S. Perez De Rosso, D. Jackson, M. Archie, C. Lao, B.A. McNamara III, Declarative assembly of web applications from predefined concepts, in: Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, in: Onward!, Association for Computing Machinery, New York, NY, USA, 2019, pp. 79–93, http://dx.doi.org/10.1145/3359591.3359728.

[35] JSON Forms, 2021, https://jsonforms.io/ (Accessed: 2021-04-18).

[36] D.M. Groenewegen, Z. Hemel, L.C. Kats, E. Visser, WebDSL: A domain-specific language for dynamic web applications, in: Companion To the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, in: OOPSLA Companion '08, Association for Computing Machinery, New York, NY, USA, 2008, pp. 779–780, http://dx.doi.org/10.1145/1449814.1449858.

[37] S. Oney, B. Myers, J. Brandt, ConstraintJS: Programming interactive behaviors for the Web by integrating constraints and states, in: Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology, UIST '12, Association for Computing Machinery, New York, NY, USA, 2012, pp. 229–238, http://dx.doi.org/10.1145/2380116.2380146.

[38] T. Felgentreff, A. Borning, R. Hirschfeld, Specifying and solving constraints on object behavior, J. Object Technol. 13 (4) (2014) 1:1–38, http://dx.doi.org/10.5381/jot.2014.13.4.a1.

[39] F. Drewes, H.-J. Kreowski, A. Habel, Hyperedge replacement graph grammars, in: Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations, World Scientific Publishing Co., Inc., USA, 1997, pp. 95–162.

[40] W.L. Miranker, A. Winkler, Spacetime representations of computational structures, Computing 32 (2) (1984) 93–114.

[41] E. Burrows, M. Haveraaen, A hardware independent parallel programming model, J. Log. Algebr. Program. 78 (7) (2009) 519–538.

[42] E. Burrows, M. Haveraaen, Programmable data dependencies and placements, in: Proceedings of the 7th Workshop on Declarative Aspects and Applications of Multicore Programming, 2012, pp. 31–40.