Towards Reusable GUI Structures

Knut Anders Stokke
knut.stokke@uib.no
University of Bergen
Bergen, Norway

Mikhail Barash
Mikhail.Barash@uib.no
University of Bergen
Bergen, Norway

Jaakko Järvi
jaakko.jarvi@utu.fi
University of Turku
Turku, Finland

Abstract

Graphical user interfaces present data as structures (lists, trees, grids). Convenient features to manipulate these structures are tedious to implement. We are working towards a GUI programming approach, where concise specifications of structures give rise to full-fledged GUIs with a complete set of structure manipulation features.

CCS Concepts: • **Software and its engineering → Graphical user interface languages**.

Keywords: GUI specification, reuse, HotDrink

ACM Reference Format:
Knut Anders Stokke, Mikhail Barash, and Jaakko Järvi. 2023. Towards Reusable GUI Structures. In *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '23), October 22–27, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3618305.3623611

1 Introduction

Graphical user interfaces (GUIs) are a means to observe and modify data. This data is organized into structures: tables, lists, trees, and graphs. Such data structures come with well-established *APIs* (Application Programming Interface), standard operations for manipulating them, which programmers can readily use. They do not have to spend time on implementing element insertion, removal, or reordering operations. GUIs, views to data, tend to be similarly structured to tables, lists, trees, and (sometimes) graphs. The operations to manipulate data via these views are, however, not well-established and structures that appear in GUIs do not come with out-of-the-box APIs. Instead, programmers must spend great effort to implement element insertion, removal, and reordering features—or inconvenience users by not doing so.

The out-of-the-box APIs are not available because GUI views are more than passive repositories of data: there are explicit and implicit dependencies amongst the GUI widgets that comprise the views. These dependencies must be maintained whenever users interact with the GUI [2, 3] and modify the data structure it represents. This is a non-trivial task especially when the dependencies are multi-directional, when the direction of how data propagates in a GUI depends on how the user interacts with the GUI [1].

The goal of our work is to show that APIs to manipulate structures in GUIs do not have to be programmed by hand. Given a concise specification of a structure that appears on a GUI and the patterns of dependencies amongst its elements, the operations for manipulating that structure can be generated and packaged for the user as convenient GUI features. Concretely, this specification is written in a *domain-specific language* (DSL), which is then transpiled into a host language (TypeScript). Our project is work-in-progress.

2 DSL for Structures in GUIs

Our DSL lets programmers specify data structures with complicated dataflows and generate GUIs from this specification; operations that modify these structures and simultaneously update the dataflow connections come for free. At its core, the DSL describes multi-way dataflow constraints, which are relational constraints on variables expressed as a set of *constraint satisfaction methods* [4], functions that enforce the relation when executed. These constraints are combined into a constraint system, and whenever users invalidate a constraint by changing a variable, a *constraint system solver* propagates the impact of the change to the rest of the system [5]. The constraint system variables are called *GUI variables* in our DSL.

Widgets in our DSL represent data fields that users can interact with (e.g., textual and numerical input boxes, check boxes). A widget has a model, which is a set of GUI variables, and a view that specifies how the widget is rendered. For simple widgets the model can be one GUI variable, but multiple GUI variables may be necessary to control more complicated widgets, such as a slider that has a minimum, maximum, and current value, and a step size. GUI variables can be part of dataflow constraints. A widget can also have a *two-way binding* between its model and view, consisting of functions for sending data between the view and GUI variables.

Components in the DSL are composed of widgets, other (sub-)components, data structures, and GUI variables. While the first three types are rendered to the view, GUI variables are hidden and serve as intermediate variables in a

SPLASH Companion '23, October 22–27, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0384-3/23/10.
https://doi.org/10.1145/3618305.3623611

dataflow connection. Dataflow connections may be specified at component-level on any of the component's parts.

The data structures supported in the DSL are lists, grids, and trees. A *list* is a sequence of components (typically) laid out either vertically or horizontally in the view. Lists provide functionality to the end-user for adding, removing, and moving list elements. GUI developers may specify dataflow between components connecting, for example, adjacent components in a list. The GUI variables of a list element can thus have dependencies on their surrounding elements to express rules such as "the departure date of every flight leg should not precede the arrival date of the flight leg above". Dataflow connections between variables in the list and variables outside the list are also possible, e.g., from the list's elements' GUI variables to a variable that maintains some aggregate value of those GUI variables.

A *grid* specified in our DSL is laid out as a table, and has a list of row header components, a list of column header components, and a cell component for the intersection of every row and column. The row and column header lists provide the same functionality as regular lists, allowing end-users to insert, remove, and move grid rows and columns. One can specify dataflow connections that are repeated for every row or for every column, similarly to dataflow connections on lists, or for every cell in the grid.

We use the grid data structure to demonstrate our DSL's capabilities. The specification in Listing 1 shows a tabular overview of `Students` and their `Assignments` and `Grades`. The grade of every student is calculated from their editable assignment scores (lines 8–9), and for the last row (`x`) we compute the average score of each assignment (lines 10–11). Finally, line 7 computes the average grade, displayed in the bottom-rightmost cell, as shown in Figure 1. We stress that from this specification a fully functioning grid GUI is generated, one that lets the user add, remove, and reorder students and assignments. The specified constraints are guaranteed to be maintained after such structural modifications.

```
1  grid<(stds: list<Student>, x: Student)
2      * (asgs: list<Assignment>, g: Grade)>
3    where {
4      (Student, Assignment): StudAsg
5      (Student, Grade): StudGrade }
6    with constraints {
7      avg((stds, g).value, (x, g).value);
8      for every s in stds {
9        calcGrade((s, asgs).score, (s, g).value)
       };
10     for every a in asgs {
11       avg((stds, a).score, (x, a).value) };
12   }
```

Listing 1. A fragment of a grid specification in our DSL.

Trees in the specification have two types of components, *tree components* and *leaf components*, where a tree component contains a list of tree and leaf components. Leaf components

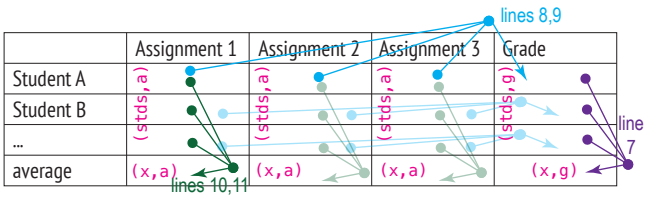

Figure 1. Dataflow in the grid specified in Listing 1.

do not contain any further components. A common use case for trees in GUIs is for nested grouping of elements, such as files, mails, and tasks. There are several useful ways to lay out trees in a user interface; all of these should be provided by the specification. The specification supports dataflow connections that make sense for trees. One example is connections between a parent node and its children, enabling parent nodes to collect data from the child nodes.

3 Discussion

The premise of our work is that the "data-structure nature" of structures that appear in GUIs is not explicit in contemporary GUI programming. If it were, commonalities between different GUIs would become more apparent and easier to take advantage of: more GUI features could be implemented in reusable libraries. The goal of our DSL is to expose these commonalities, letting the programmer specify GUIs by describing how the data is structured and what its internal dependencies are, and generate fully working GUIs from these dependencies.

Our DSL is currently quite experimental. It allows for generating list and grid structures, trees are on the drawing board. The generated code performs updates inefficiently, recomputing all data dependencies after every structural modification; we do not yet analyze the reach of an operation to avoid unnecessary updates. We have implemented small GUI examples, demonstrating that very concise specifications can give rise to feature-rich correctly functioning GUIs with complex dataflows. We need further work with more complex—and a wider variety of—examples to gain more confidence that the approach is flexible enough for practical GUI programming.

References

[1] J. Järvi et al., *Algorithms for user interfaces*, SIGPLAN Not. 45:2. 2009.
[2] K. A. Stokke et al., *A domain-specific language for structure manipulation in constraint system-based GUIs*, J. Comp. Lang. 74. 2023.
[3] K.A. Stokke et al., *The Ultimate GUI Framework: Are We There Yet?*, EVCS 2023.
[4] M. Sannella, *Skyblue: A Multi-way Local Propagation Constraint Solver for User Interface Construction*, ACM. 1994.
[5] B. Vander Zanden, *An Incremental Algorithm for Satisfying Hierarchies of Multiway Dataflow Constraints*, ACM Trans. Program. Lang. Syst. 18:1. 1996.

Received 2023-08-15; accepted 2023-08-30