ReverseORC: Reverse Engineering of Resizable User Interface Layouts with OR-Constraints

Yue Jiang
Max Planck Institute for Informatics
Saarbrücken, Germany
yuejiang@mpi-inf.mpg.de

Wolfgang Stuerzlinger
School of Interactive Arts + Technology (SIAT), Simon Fraser University
Vancouver, Canada
w.s@sfu.ca

Christof Lutteroth
Department of Computer Science, University of Bath
Bath, UK
c.lutteroth@bath.ac.uk

Figure 1: Reverse engineering of the MS Word "ribbon" toolbar. ReverseORC samples the user interface (UI) at different sizes and reconstructs parsimonious layout specifications for each size. It then detects changes between the layout specifications using a novel diff algorithm for layouts, and matches the changes with corresponding layout patterns to reconstruct a UI with the same resize behaviours as the original. We visualize the overall quality of reconstruction at different sizes in an error map by color-coding structural error (shades of yellow), transition error (blue/green), and 'fault lines' (black) indicating potentially inconsistent behaviors.

ABSTRACT

Reverse engineering (RE) of user interfaces (UIs) plays an important role in software evolution. However, the large diversity of UI technologies and the need for UIs to be resizable make this challenging. We propose ReverseORC, a novel RE approach able to discover diverse layout types and their dynamic resizing behaviours independently of their implementation, and to specify them by using OR constraints. Unlike previous RE approaches, ReverseORC infers flexible layout constraint specifications by sampling UIs at different sizes and analyzing the differences between them. It can create specifications that replicate even some non-standard layout managers with complex dynamic layout behaviours. We demonstrate that ReverseORC works across different platforms with very different layout approaches, *e.g.*, for GUIs as well as for the Web. Furthermore, it can be used to detect and fix problems in legacy UIs, extend UIs with enhanced layout behaviours, and support the creation of flexible UI layouts.

CHI '21, May 8–13, 2021, Yokohama, Japan

ACM ISBN 978-1-4503-8096-6/21/05.
https://doi.org/10.1145/3411764.3445043

CCS CONCEPTS

• **Human-centered computing** → **User interface toolkits**.

KEYWORDS

ORC Layout; reverse engineering; constraint-based layout; adaptive user interface; resizable user interface

ACM Reference Format:
Yue Jiang, Wolfgang Stuerzlinger, and Christof Lutteroth. 2021. ReverseORC: Reverse Engineering of Resizable User Interface Layouts with OR-Constraints. In *CHI Conference on Human Factors in Computing Systems (CHI '21), May 8–13, 2021, Yokohama, Japan.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3411764.3445043

1 INTRODUCTION

Modern user interfaces (UIs) have become increasingly flexible. They use sophisticated layouts that can adapt to different sizes and orientations. For example, responsive web layouts [40] enable designers to create web UIs that work on large desktop screens, small tablets, and tiny mobile devices by rearranging and adapting the UI. Similar approaches are used for mobile UIs [58, 72] and sometimes even for desktop UIs. The UIs and their layouts are created using UI toolkits [25, 26, 45] and UI builders [60, 73], which facilitate the efficient creation and editing of common layouts and support an iterative design process. However, available UI toolkits, builders and supported layouts are numerous and constantly evolving, leading to a wide diversity of different layouts such as grid, flow, group, stack, tile, flexbox and constraint-based layouts.

It is challenging to change an existing UI if its source code or specification is not available. Even if a specification is available, it is usually tied to the UI toolkit that was used to create the UI, and the diversity of UI toolkits and layouts makes it hard to understand and use such a specification. The problem of reconstructing an existing UI for further development is called UI *reverse engineering* (RE). It is known to be difficult but often necessary as software and devices evolve and new UI toolkits and platforms need to be supported. For example, developers may want to modernize a legacy UI to benefit from novel technologies – a use case where it is quite common that source code is not available, hard to modify, or an equivalent layout API may not exist. An illustrative case here is porting desktop GUIs to smartphones or vice versa, or even to/from webpages. It is very challenging to reverse engineer a UI so that its features and behaviours are consistent across different toolkits and platforms; so developers usually spend a lot of time trying to understand a specification and often end up manually re-implementing large portions of the UI [18, 23, 52].

In order to ease the burden of UI RE, many automated RE tools have been proposed. By using automated RE tools, specifications of GUI elements, layouts, and application behaviours can be extracted and modified independently of their underlying implementations and platforms. Previous works on reverse engineering UIs focus on detecting components in the interface [8, 42, 64], migrating UIs from one platform to another [17, 37, 41, 43, 44, 56, 57], and/or performing input and output redirection [9, 10, 64, 65]. Previous works have shown that RE tools can reconstruct UI layouts that look similar to the originals, and can then generate implementations of the UI for other UI platforms and toolkits. However, while UI layout has evolved, RE tools have not kept pace with modern UIs: they cannot currently capture the complex resizing behaviours that have become commonplace for the web, on mobile devices, and even many desktop UIs.

This paper presents ReverseORC, a novel prototype that is able to reverse engineer UI layout specifications based on a UI's resize behaviors. Given only widget placements for different UI sizes of an existing GUI, ReverseORC identifies how layout behaviours are encoded in the UI and generates a corresponding layout specification. The new layout specification is expressed using ORC Layout [30], an approach for constraint-based layouts based on OR-constraints (ORC). ORC Layout is a powerful tool that allows designers to express modern resizable UI layouts in a mathematical, platform-independent manner, as constraint optimization problems. It unifies flow layouts and conventional constraint-based layouts to represent a large variety of layouts for desktop, web and mobile platforms. We build ReverseORC on ORC Layout as this is one of the most flexible layout specification mechanisms that does not involve programming.

ReverseORC automatically extracts specifications of how a UI is laid out for different UI sizes. It determines which UI size samples are necessary to infer an equivalent ORC layout specification. Although it is not possible in general to reverse engineer an arbitrary layout algorithm solely from examples for its output, ReverseORC is able to detect common layouts such as grids and flow. Additionally, it is able to detect advanced patterns describing dynamic changes in a layout (ORC patterns), such as widgets shifting or disappearing when the UI is made smaller. ReverseORC is designed to generate parsimonious specifications, i.e. specifications that are sufficiently rich to capture the desired behaviour, but not more complex than necessary. This makes ReverseORC's output easier to understand for a human designer, so that they can potentially build on it later. Only if ReverseORC cannot identify a pattern in the observed changes, then it uses OR constraints to combine the specifications of the respective layouts.

We demonstrate that ReverseORC can be applied to UIs on different platforms, such as desktop and web UIs, reconstructing platform-independent specifications for a wide range of UI technologies. Furthermore, we support a variety of use cases based on the generated ORC layout specifications: Many existing GUI layouts are static or cannot fit a large range of screen sizes adaptively, *e.g.*, from smart watches and smartphones to desktop environments. Designers can specify desired adaptations of such a layout by example, and let ReverseORC reconstruct an appropriate flexible layout specification. In a similar manner, designers can use ReverseORC to create new resizable layout specifications from scratch. Furthermore, ReverseORC allows designers to fix bad layout behaviors by modifying the generated ORC layout specification. In summary, ReverseORC lifts the level of abstraction of the layout specification process by allows designers to create and edit sophisticated flexible layout specifications by example.

Novelty. In contrast to existing GUI reverse engineering approaches [17, 37, 41, 43, 44, 56, 57], ReverseORC is able to reconstruct the *dynamic* resizing behaviors of a GUI. Extending previous work that identified static layout components, it detects advanced layout patterns such as optional and shifting widgets and specifies their behaviors. It is platform and toolkit independent, which enables reuse of layouts across applications and platforms. Finally, it allows designers to specify the resize behavior of UIs by example. In particular, we demonstrate the following contributions:

(1) A novel method for identifying and reverse engineering dynamic layout behaviors for different platforms, only by sampling different layout sizes for an existing UI.

(2) A novel method for detecting the differences between layout specifications.
(3) A novel method of specifying and editing resizable layouts by example.
(4) Validation of our approach based on real-life layouts, including GUI layouts, *e.g.*, the Microsoft Word Ribbon, and web layouts, *e.g.*, the BBC News website, as well as demonstrating that ReverseORC can reverse engineer layouts based on a very small number of exemplars.

2 RELATED WORK

2.1 Resizable UI Layout

Due to the large diversity of existing computing devices, which vary in their screen sizes and aspect ratios, and users' different personal viewing preferences, it is important that applications support resizable UI layouts. Layout models are widely used to specify resizable UI layouts, and layout managers then generate the layout results based on the specifications. Early approaches proposed simple layout models, such as group, grid, table, and grid-bag layouts [45, 47]. Object-oriented models like Amulet [48] combined properties of flow and grid layouts. Modern GUI layout models are mostly constraint-based [38, 74] and used together with UI builders, which can create layout constraints based on direct manipulation [32, 60, 69, 73].

Jiang et al. [30] proposed ORC Layout, an approach for constraint-based layouts based on OR-constraints (ORC). An OR-constraint is a disjunctive constraint, where only one disjunctive part needs to be true. ORC Layout unifies flow and conventional constraint-based layouts through adding OR-constraints to standard soft/hard linear constraint systems. ORC Layout specifications also enable the use of ORC design patterns, which enable designers to create a large variety of flexible layouts that work across different screen sizes and orientations. ORC Layout is a powerful, high-level layout specification method; it enables users to describe layouts with dynamic behaviors that adapt to screens with very different sizes, orientations, and aspect ratios, using only a single layout specification. ORCSolver [31] is a novel solving technique to efficiently solve ORC Layout specifications. ReverseORC uses ORC Layout to express the captured dynamic layout behaviors.

Previous work also investigated resizable web layouts. Chen et al. [7] presented a page-adaptation technique that splits a web page into smaller blocks to adapt pages for small screen devices. Xie et al. [70] proposed a novel document representation dynamically adapting screen sizes. Domshlak [13] enabled personalized presentation of web page content. Constraints can also be used to specify the desired layout of a web page, e.g. Borning et al. [4] proposed a constraint-based web system enabling both the author and the viewer to define page layout constraints. Hosobe [22] introduced an algorithm to solve hybrid systems of linear constraints and one-way constraints to handle web document layouts efficiently.

2.2 Customizing User Interfaces

Researchers have proposed several approaches that can be used to modify a GUI if it is not automatically adapted to the user's requirements or if the adaptation is sub-optimal, *e.g.*, when using a GUI on a device with a smaller size. For traditional GUIs, Edwards et al. [14] and Olsen et al. [53] proposed to modify interfaces by replacing drawing objects and intercepting API commands in applications with specific toolkit implementations. WinCuts [66] enabled window subdivision with a copy-paste method to configure input/output redirection. Mudibo [28] used input/output redirection to generate windows with multiple alternative positions, and allowed users to choose a desired one. User Interface Façades [64] detected all widgets and their hierarchy through an accessibility API, enabled widget replacement, and presented advanced customization of runtime interaction behaviour.

Previous research on web UI customization was mostly based on a structured presentation, the Document Object Model (DOM). ChickenFoot [3], CoScripter [34], and Koala [36] automated, customized, and integrated web applications. Clip, Connect, Clone [16], d.mix [19], and Vegemite [35] introduced end-user mash-up methods between existing applications. Highlight [51] re-authored web applications on mobile interfaces.

As all UIs are instantiated as pixels, previous work widely explored pixel-level interpretation to enhance UIs. Pixel-based approaches have been proposed to access data [55], record the actions performed by users [61], translate input and output into different forms [1], improve target detection in accessibility APIs [27], perform visual manipulation [75], and event management [76]. ScreenCrayons [54] enabled document and visual annotation. Sikuli [6, 71] supported UI testing by writing visual test scripts. Genie [65] reverse engineered underlying commands to enable users to engage with web applications via different input modalities. Hurst et al. [27] presented improved target boundary detection based on the combination of an accessibility API and pixel-based methods. ReverseORC is a platform- and framework-independent system enabling customization for both GUI layouts and web layouts.

Table 1: Overview of reverse engineering approaches ('+' denotes full and '∼' partial support for certain layouts).

	Basic Layouts	Flow Layouts	Dynamic Topology
Most previous RE work	+		
Constraint-based RE [37]	+		
Model-driven RE [56, 57]	+	∼	
Expresso [33]			∼
InferUI [2]	+		
ReverseORC	+	+	+

2.3 Reverse Engineering

UI reverse engineering is widely used to migrate applications from one platform to another. Moore [42] presented a rule-based detection approach for partially automating the process of reversing engineering legacy applications. Staiger [62] analyzed the source code, identified widgets, and reconstructed the GUI tree. MORPH [41, 43] proposed a model-oriented re-engineering process for migrating character-based legacy UIs to GUIs. REMAUI [50] was a pixel-based approach that automatically reverse engineered mobile application UIs. None of the above approaches yielded resizable layout information.

Reverse engineering has been used as a way to perform GUI customization. UI Façades [64] enabled users to replace widgets and change application behaviors for an existing application at runtime through an accessibility API-based approach. Prefab [8–12] was a pixel-based approach that provided a tree structure to interpret content and hierarchy [11]. Both approaches identify interface elements and allow the user to add interactive enhancements to a GUI [8, 12, 64]. However, none of these approaches allowed users to modify the layout itself. Instead of using pixel-based interpretations of a UI for reverse engineering [12] or migrating a UI directly between different platforms [17, 59], our approach detects layout behaviors and generates standard ORC Layout specifications to facilitate UI development and customization. Similar to ReverseORC's layout structure reconstruction, InferUI [2] infers constraints to describe a layout from UI exemplars. Yet, InferUI generates only linear constraints, which maintain relative mutual alignments of widgets but can only express a single topological arrangement. In contrast, ReverseORC infers OR-constraints, which can express dynamic topological layout changes such as flow, optional widgets, and alternative positions.

Lutteroth [37] reverse engineered GUI layouts to recover higher-level constraint-based specifications [39] and to generate layouts that are resizable. Sánchez Ramón et al. [56, 57, 59] proposed a model-driven approach to reverse engineer legacy GUIs by capturing the visual arrangement of elements in the layout and produced GUI models with that explicit layout. While these approaches [56, 57] can capture common layout containers in a hierarchical manner, ReverseORC is also able to reconstruct a platform-independent specification of dynamic UI changes, such as optional widgets or widgets that change position across the layout hierarchy to accommodate changes in screen space, which cannot be expressed with common layout containers. The above approaches were only able to deal with simple layout behaviors such as grid arrangements, but could not deal with layouts that included dynamic layout changes such as flows, shifting widgets or optional widgets.

Reverse engineering is also useful for web layouts. Moore et al. [44] used the MORPH technique [41, 43] to re-engineer legacy information systems to operate on the web. CELLEST [63] demonstrated a process for migrating legacy GUIs to web-accessible platforms. Gerdes [17] proposed a method to migrate Windows applications to Visual Basic .NET, based on runtime traces. VAQUISTA [67] reverse engineered the presentation model of a web page to generate equivalent GUIs for other platforms. VIPS [5] presented an approach for web content structure analysis based on visual representation. Similar to ReverseORC's exemplar-based layout design, Expresso [33] allows designers to specify samples of a web UI at different sizes. Expresso then either linearly interpolates widget positions and sizes between the given UI sizes ('keyframes'), or lets them jump discontinuously, as specified by the designer. In contrast to ReverseORC, Expresso does not infer behavioral UI layout patterns dynamically. For example, if widgets should flow onto a new line, the designer would have to specify keyframes for every possible line break in Expresso. To the best of our knowledge there is no reverse engineering approach for UI layouts that can extract a UI's dynamic resize behaviours. Table 1 shows a comparison of the capabilities of different reverse engineering approaches.

3 OVERVIEW

Our ReverseORC approach first extracts widget information from the layout through accessibility APIs. Then, it uses a grid search to sample and resize the layout through setting different window sizes (Figure 2 a). It constructs a layout tree for each sampled layout (Figure 2 b). ReverseORC then tracks all differences between layout trees of neighboring layouts during the sampling process and generates corresponding layout differences. Based on these layout differences, ReverseORC then infers overall layout behaviors and patterns, and constructs a corresponding ORC layout specification, enabling later modification and customization (Figure 2 c).

3.1 Usage Scenarios

ReverseORC fits into standard software development practice and has many practical applications. Some typical usage scenarios are:

(1) Developers initiate automatic UI sampling with a platform-specific tool: For desktop UIs, developers run the app to reverse engineer. Our tool then uses OS calls to set the UI window size and accessibility API calls to query widgets and their properties. For web UIs, developers use a tool with an embedded browser, instrumented to resize and extract widgets after the UI has been loaded. Similarly, for mobile UIs we use an emulator, with emulator calls to set the screen size and extract widgets. For each UI sample, all widgets and their properties are stored in a platform independent format. Previous work has demonstrated such approaches.
(2) Layout structure reconstruction, difference detection, and ORC specification generation are performed automatically based on the UI samples with our platform-independent tool. The tool visualises the quality of the reverse engineered UI (see 6.2) and allows developers to display it at specific sizes by clicking on points on the error map. Developers can adjust the results and fix bad layout behaviors by modifying and adding UI samples (see 7.3), or editing the ORC specification directly using the ORC Editor [30, 31].
(3) ORC UI specifications can then be used directly by running them on a platform-specific implementation of the ORC Solver, which can run on desktop and mobile platforms as well as the web[1].

4 USER INTERFACE SAMPLING

To reverse engineer a GUI, we follow the common approach of first detecting the widgets of the UI, and then reconstructing the layout of the widgets using ORC Layout as abstraction model. Subsequently, we transform the reconstructed specification to its target form, generating a new UI for the desired platform. As discussed above, previous works only reconstruct lower-level UI specifications that ignore the more abstract aspects of UI layout during this process. By sampling an existing UI at different layout sizes, ReverseORC can identify and reverse engineer both GUI layout and web layout behaviours. It first extracts widget information from the layout through accessibility APIs. Then, it uses a grid search to sample and resize the UI layout by setting different window or screen sizes. ReverseORC keeps track of any differences between

[1] github.com/cpitclaudel/z3.wasm

Figure 2: Overview of the different stages of the ReverseORC approach: a) UI sampling by setting different window sizes (horizontal lines are divisors that define *Row* layout containers); b) layout tree structure construction for each sample; and c) difference detection between layout trees and layout pattern inference.

neighbouring layouts during the sampling process, and is then able to reconstruct an abstract layout specification based on the way the layout changes depending on its size.

4.1 Widget Extraction

Similar to UI Façades [64], our approach extracts widget information of UIs through an accessibility API. An accessibility API provides a structured representation analogous to the Document Object Model (DOM). Compared to pixel-based approaches [8] or computer vision recognition-based approaches [50], accessibility APIs directly access the underlying data of a UI, which avoids the potential for recognition errors. In addition, accessibility APIs can access information that is not visible or not obtainable by analyzing raw pixels, *e.g.*, widget identities. We still acknowledge that pixel-based approaches could be used as an alternative mechanism in ReverseORC, albeit at the price of an increased risk of layout recognition errors.

To extract the widget information ReverseORC needs, we traverse the structured representation through the accessibility API. Under the assumption that the bounding box of each widget is rectangular, for each widget w in the layout, we retrieve its unique identifier ($w.id$), size ($w.width$, $w.height$), and coordinates for its top-left corner ($w.left$, $w.top$). Some accessibility APIs provide more information about a UI, including information not only about the widgets but also about the layout managers used. For example, it is generally possible to access the full DOM of a web UI. However, ReverseORC does not use this information for the following reasons: 1) Layout information is not always available, *e.g.*, some desktop UIs do not provide it. 2) There are too many layout managers to understand the layout behavior of a UI just from the DOM, so DOM layout containers are often like black boxes. 3) Layout behaviors are often described at least partly programmatically rather than in the DOM, *e.g.*, using JavaScript code; therefore they cannot be inferred from the DOM alone. And 4) even if we could interpret a DOM description of a UI layout, DOMs are often much more complicated than they need to be. For example, many complex web apps use large numbers of nested DIV elements, confounding aspects of layout and functional application design. One of the aims of ReverseORC is to provide a parsimonious layout representation, *i.e.*, a representation that avoids unnecessary complexities. This is achieved by analysing not how developers have specified layouts, but by analysing what layouts actually looks like, in the simplest terms possible.

4.2 Grid Search

We use an adaptive grid search approach to obtain a representative set of different layout exemplars by resizing the window or a virtual screen. A brute force method to thoroughly analyze a layout would be to sample as many exemplars as possible. However, in practice, it can be expensive to resize the layout to all potential sizes, and it would create unnecessary work for the later reverse engineering stages. Thus, it is best to minimize the number of queries by taking advantages of the continuous nature of UI layout: layout changes occur incrementally, as it would otherwise confuse the user. If two sampled layout exemplars have the same structure or their variance matches layout behaviours we have already detected, then there is no need to subsample further and to explore more exemplars in the range between the sizes of these two layout exemplars. In this case we (very likely) have already identified all the behaviors in this range and are unlikely to get more information by further subsampling.

We chose to perform an adaptive grid search to sample UI layout, as layout sampling is a two-dimensional problem. Both the width and height of a UI are likely to affect its layout, with UIs often assuming different layouts for different sizes and aspect ratios. We start with the extreme window sizes (minimum and maximum) and sample new layout sizes based on a binary grid search. We define the maximum size of a layout as the biggest screen size we would like to support, and the minimum size as is defined by the layout, i.e., the minimum size that the UI can be set to. During the search process, if two sampled layout exemplars $L1$ and $L2$ have the same structure or their variance matches layout behaviors we

have already detected, then we stop subsampling in the window size range between these two layout exemplars. Otherwise, we subsample depending on their sizes. If they have the same width but different heights, we subsample a layout exemplar with the same width and the middle height of the two. Analogously, if they have the same height but different widths, we subsample a layout exemplar with the same height and the middle width of the two. If both width and height are different, we subsample three layout exemplar with 1) middle width and middle height, 2) same width as $L1$ and same height as $L2$, and 3) same height as $L1$ and same width as $L2$, respectively.

We show a sampling example in Figure 3, referring to the layout exemplars as (Min), (Max), (1), (2), etc. We start with the extreme layout sizes, *i.e.*, the minimum size (Min) shown at the top left and the maximum (Max) at the bottom right. According to the subsampling rules, based on (Min) and (Max), we subsample layout exemplars (1), (2) and (3). To minimize the number of subsampling exemplars, we first subsample between two exemplars with the same height or width, and perform further subsampling along the diagonals between exemplars only if both variations in height and width show changes in the layout structures. For example, we subsample layout (4) between (Max) and (1). Widget 3 disappears in (4), so we continue to subsample between (Max) and (4) to detect the point of its disappearance, stopping the subsampling once the size difference between two exemplars is small. The structural difference between (Max) and (3) is the same as the difference between (Max) and (4), *i.e.*, widget 3 disappears. As we have already subsampled between (Max) and (4), we do not subsample further between (Max) and (3). We keep subsampling until we find all structural differences of the exemplars and the approximate transition points of changes.

5 LAYOUT STRUCTURE RECONSTRUCTION

In order to compare the differences between layouts, we aim to reconstruct the simplest possible specification for the structure of a layout exemplar. We use symbolic tabstop dividers to divide layouts into separated parts in order to define layout structure. Such layout structure makes it easier to compare layouts and detect the differences between layouts.

5.1 Tabstops

A tabstop is an abstraction that has been introduced in previous work on GUI layouts [21, 24, 39, 73]. It is a symbolic object in the layout used to represent the alignments of multiple widgets. Associated with the two dimensions of the plane, there are two types of tabstops: x-tabstop and y-tabstop. An x-tabstop represents a position on the x-axis and correspondingly for a y-tabstop. Tabstops are in effect variables defining horizontal (y-tabstops) or vertical (x-tabstops) grid lines. The combination of x-tabstops and y-tabstops in a GUI forms a grid controlling how widgets are aligned in the GUI. Each widget w has four tabstop variables ($w.left$, $w.right$, $w.top$, $w.bottom$) that delimit the area it occupies. Similarly, a layout L itself has four tabstop variables $L.left$, $L.right$, $L.top$ and $L.bottom$ that define its boundaries, which is typically called the window (or panel) size.

The main advantage of using tabstops is that in a constraint-based layout system, if some widgets share a boundary, can just add a tabstop to the specification and then have all the corresponding widgets refer to that tabstop instead of adding separate alignment constraints for each widget. This approach makes it easier to maintain and modify the resulting constraint system. Whenever we need to change the alignment of the widgets sharing a tabstop, we just need to change the constraints relating to that tabstop, and then all the corresponding widgets will be positioned accordingly. For each layout L, we define tabstops through two functions $xtabs()$ and $ytabs()$ that map from positions in the GUI to tabstop variables in the layout: $xtabs$ is a function mapping from x-coordinates to x-tabstops, and $ytabs$ mapps y-coordinates to y-tabstops (See Appendix A for details about tabstop creation).

To guide layout reconstruction, we call a tabstop a *layout divider* if it is a clean cut dividing the layout into two parts without crossing any widget in the layout. To reconstruct the containment hierarchy of a UI, the concept of layout dividers is applied recursively on the sublayouts contained in a layout. For example, in Figure 4 the orange lines are the vertical layout dividers of the overall layout, and the green lines are horizontal layout dividers of sublayouts. Figure 5 shows two examples of subdivision results. For a horizontal layout divider, all the widgets in the layout are either above it or below it, and analogously for vertical layout dividers (See Appendix B for details about tabstop layout divider detection).

5.2 Reconstruction Algorithm

Our layout structure reconstruction algorithm uses the same principles as the XY-Cut algorithm [29, 49] but works at a higher level of abstraction. Rather than segmenting an image based on gaps, we consider widget boundaries directly and we remove cuts if this allows us to simplify the X-Y structure.

We define layout structure using *Row* and *Column* layout containers. Two widgets belong to the same *Row* if they are located between the same two horizontal layout dividers, and analogously for *Column*. The resulting layout structure is a nested *Row* and *Column* structure. We reconstruct the layout structure by recursively subdividing it based on layout dividers. We try horizontal subdivision (with vertical layout dividers) first as it is more common and in line with reading order. If horizontal subdivision is not possible, we process vertical subdivision analogously. We then assign the widgets to different sublayouts based on the positions of the horizontal layout dividers, and recursively use the reconstruction on each sublayout structure. If both cases are impossible, which is very rare as UIs are typically laid out using a division-based containment hierarchy, then the layout can only be described using tabstops, *e.g.*, in a pinwheel layout [74] (see Appendix C for details about layout structure construction). Figure 4 shows the visualization of the reconstructed layout structure of the MS Word "ribbon".

We aim to reconstruct the simplest possible layout structure. To avoid creating layout dividers caused by accidental alignments, we regroup widgets in multiple consecutive sublayouts and try running the algorithm recursively to simplify the resulting layout structure. We reconstruct the sublayout if we can get a simplified sublayout structure by grouping them.

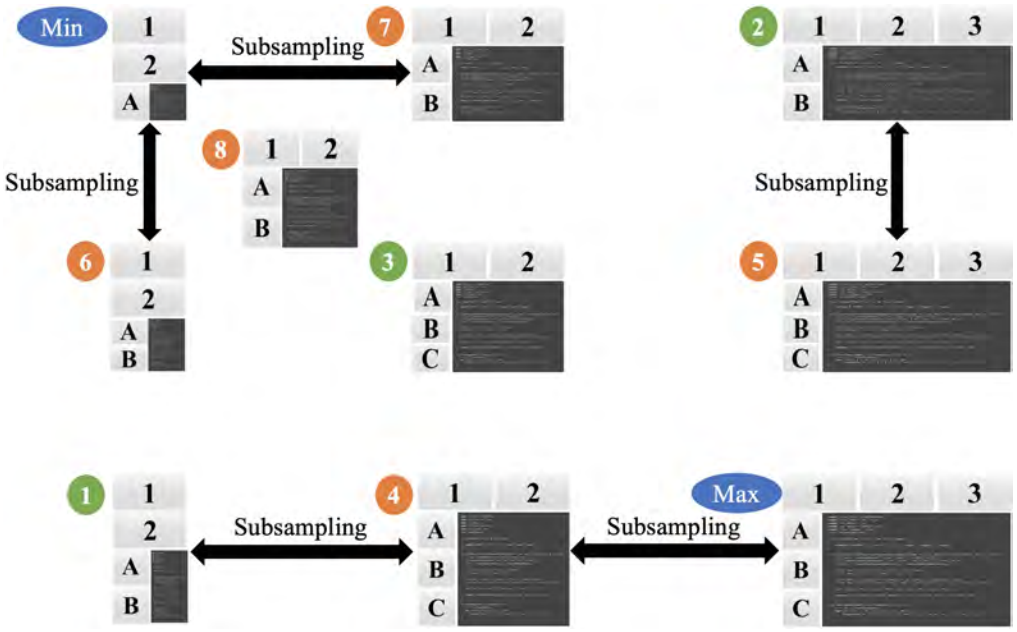

Figure 3: Grid subsampling example.

Figure 4: Visualization of the layout structure of the MS Word "ribbon" as reconstructed in Section 5.2. Horizontal lines are divisors that define *Row* layout containers and vertical lines define *Col(umn)*s.

6 LAYOUT DIFFERENCE DETECTION

ReverseORC keeps track of differences between neighbouring layouts during the sampling process and generates corresponding layout change sets. Based on the constructed layout structure, we can generate a corresponding layout specification tree where widgets are leaf nodes, and *Row*s and *Column*s are internal, non-leaf nodes. Our layout difference detection algorithm takes two layout trees as input and generates a set of edit operations that indicates the differences between the two. The set of edit operations is then used to infer layout behaviors.

Although there are many existing difference detection algorithms [15, 20, 68], they cannot easily be applied because UI layouts have different requirements than other common tree structures such as XML or source code. Previous tree difference detection algorithms usually either consider none of the tree nodes to have a unique identity, or all of the nodes to have a unique identity. However, in the case of layouts, some tree nodes (the widgets) have a unique identity and some have not (the layout nodes). We can observe the widgets from the outside, *e.g.*, through an accessibility API, and can identify them. However, we cannot reliably identify layout elements such as rows and columns as accessibility APIs usually do not deliver this information; we infer their presence only by the way the UI is structured. So in a nutshell, our difference detection algorithm must be able to work with identities for some nodes, but not others. In addition, previous difference detection algorithms often only supported the detection of deletion, insertion, and moving. For thoroughly analyzing and comparing layouts, we need more edit operations, such as whether a Row has changed to a Column. Furthermore, the computational complexity of generic tree difference detection algorithms are often quadratic. Our layout tree difference detection algorithm only takes linear time for detecting layout differences in practice.

6.1 Edit Operations

We define the following edit operations that can be applied to change a layout specification $S1$ to another specification $S2$, thus indicating the differences between them:

- *addNode*($s2$): add node $s2$ in $S2$
- *removeNode*($s1$): remove node $s1$ from $S1$
- *moveNode*($s1, s2$): move node $s1$ in $S1$ to node $s2$ in $S2$
- *replaceNode*($s1, s2$): replace node $s1$ in $S1$ with node $s2$ in $S2$
- *changeType*($s1$, *toType*): change the type of node $s1$ to *toType*
- *changeChildrenOrder*($s1$, *toOrder*): change the order of the children of node $s1$ to *toOrder*

6.2 Layout Tree Data Structure

We encode a reconstructed layout structure in a corresponding tree. Each widget becomes a leaf node, while each *Row* or *Column* becomes an internal node. For example, Figure 5 shows the layout tree for the reconstructed layout structure of the MS Word "ribbon" in Figure 4. Each node stores its properties: $widgetId$, $type$, $parent$, $children$, $pathToRoot$, $hashCode$, and $childHashCode$ (detailed descriptions of node properties are given in Appendix D). $hashCode$ is defined recursively: for leaf nodes is it $hash(widgetId)$ based on a standard hash function, and for internal nodes it is $hash(type) + 1 \times child1.hashCode + 2 \times child2.hashCode + 3 \times child3.hashCode + ...$, where $child1, child2, child3, etc.$ are children nodes of the current node. Similarly, $childHashCode$ is defined the same as $hashCode$ for leaf nodes, and as $child1.hashCode\ XOR\ child2.hashCode\ XOR\ child3.hashCode\ XOR\ ...$ for internal nodes. $hashCode$ depends on the widget identity for leaf nodes, and the structure type, children nodes and their order for internal nodes. Thus, if two nodes in two layout specifications have the same $hashCode$, then they are identical with very high likelihood. $childHashCode$ only depends on the list of children of an internal node. It can be used to find corresponding nodes even if the node type (*e.g.*, changing from *Row* to *Column*) or the order of children has changed. We define two hash tables to keep track of $hashCode$ and $childHashCode$: $hashMap$ maps the $hashCode$ of a node to the node itself, and $childHashMap$ maps from the $childHashCode$ of a node to the node itself. When we compare two layout specifications, these two hash tables are used for quickly identifying corresponding nodes in both specifications (See more details in Appendix D).

6.3 Difference Detection Algorithm

We detect the differences between two layout specification trees (*Tree1* and *Tree2*) and identify edit operations by recursively comparing corresponding lists of sibling nodes $S1$ and $S2$, with the corresponding lists containing child nodes of nodes that have already been determined to correspond. The basic idea is: we try to match nodes in $S2$ with corresponding nodes in *Tree1*. Whenever we have found a correspondence, we compare the respective nodes and record edit operations for any differences in position, type, or child order. We identify the corresponding nodes in the sibling lists and recursively apply this algorithm to the child nodes of corresponding nodes.

We first try to detect strong correspondences based on hash values to find all the nodes $s2 \in S2$ such that there is a node $s1 \in Tree1$ with the same $hashCode$ or $childHashCode$ as $s2$. We then identify edit operations based on the differences between the corresponding nodes (for details see Appendix E). We expect most $s2 \in S2$ to have a corresponding node $s1 \in Tree1$. Thus, after this step, there should only be very few remaining nodes. If we cannot find corresponding nodes for all the nodes $s2 \in S2$, we pair the remaining nodes in both trees based on their similarity. We keep pairing the remaining nodes depending on the largest number of common leaves and recursively call the algorithm on their child nodes (see Appendix F for layout difference detection details). For example, in Figure 2 c, we identify move behaviors after detecting corresponding nodes in the two layout trees and comparing the node position differences.

As most nodes typically can be easily paired based on their $hashCode$ and $childHashCode$, ReverseORC takes roughly linear time to process such nodes. Very few remaining nodes need to be paired based on similarities. Thus, the overall complexity of the layout difference detection algorithm is linear in practice for all GUI layouts we have tested.

7 ORC LAYOUT SPECIFICATION GENERATION

Based on the layout differences, ReverseORC infers layout behaviors and constructs corresponding ORC layout specifications, enabling later modification and customization. We use a pattern matching approach to find ORC Layout patterns that can describe the detected edit operations. Because layouts generally change incrementally, the set of detected edit operations usually only contains a small number of edit operations. These edit operations indicate the smallest changes in the UI and thus have a clear mapping to layout patterns, which enables us to perform precise pattern matching.

7.1 ORC Layout Pattern Matching

ORC Layout [30] is one of the most flexible layout specification mechanisms that does not involve writing code. It comes with a set of layout patterns that can be used to specify common – and also several not so common – layout behaviours. By iterating over the layout transitions that we identified earlier on in the grid search, and considering each of the change sets identified by the tree difference detection, we identify and record ORC Layout patterns that can elicit the observed changes. We start with the specification of the UI at its maximum size, and iterate 'inwards' (right and up) over the samples and their change sets towards a UI's minimum size, i.e., in a way that describes a gradual change from the maximum to the minimum layout. For example, in the grid from Figure 3, change sets are considered in the following order: (Max) to (4) to (1), (Max) to (5) to (3) to (6), (3) to (8), (5) to (2) to (7) to (Min).

In each iteration, we match a layout pattern to the respective change set and generalise the layout specification to include the respective pattern. The mapping between the edit operations in the change sets and the patterns is fairly direct, so patterns can be found by iterating over the edit operations and testing each pattern for applicability in a rule-based manner. If a pattern is applicable, we adjust our ORC layout specification to include the respective pattern. The most common edit operations and their associated patterns are as follows:

removeNode(s1): If $s1$ is a leaf node, then $s1$ is an optional widget and we change the specification to mark it as such ("$s1$ is either there OR not"), using the current layout area (width × height) as penalty. As a result, a widget that disappears only when the layout gets small will have a small penalty, and the layout solver will implement this expected behavior. If $s1$ is a non-leaf node, this could be a knock-on effect of a flow layout with a *Row* or *Column* disappearing. We therefore check whether all the children of $s1$ have been moved away with corresponding *moveNode* operations. If that is not the case, $s2$ is marked as an optional sublayout. Otherwise, we ignore this operation as it will be handled by a different rule. *addNode(s2)*: This is the inverse case to *removeNode(s1)* and is handled analogously. If $s2$ is a non-leaf node, this could be a knock-on

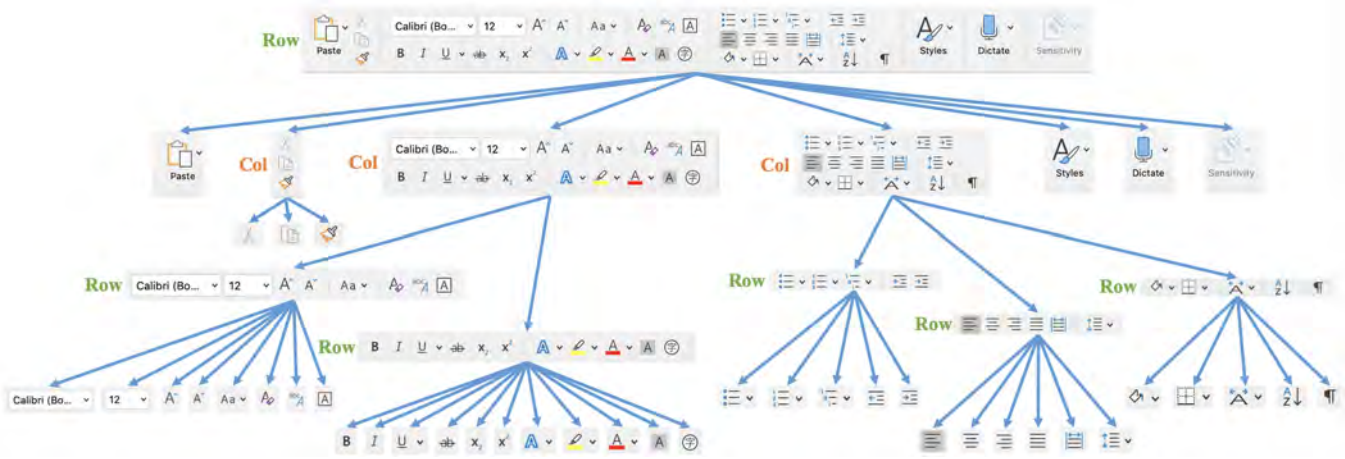

Figure 5: Layout tree for the reconstructed layout structure of the MS Word "ribbon" in Figure 4.

effect of a flow layout with a new *Row* or *Column* appearing, so we test this first. *moveNode(s1, s2)*: If one or more consecutive nodes at the end of one Row / Column are moved to a (possibly new) adjacent Row / Column, then we merge Rows / Columns into a Horizontal / Vertical Flow. Otherwise, $s1$ has an alternative position at the location where $s2$ is, and we specify this by using ORC Layout's alternative position pattern ("$s1$ is either at position 1 OR at position 2"). *replaceNode(s1, s2)*: $s1$ and $s2$ are alternative nodes, so we use ORC Layout's alternative layout pattern ("there is either $s1$ OR $s2$ at that location"). *changeType(s1, toType)*: $s1$ is marked as a pivot sublayout as a Row has changed to a Column or vice versa ("$s1$ is either a *Row* OR a *Column*"). *changeChildOrder(s1, toOrder)*: $s1$ has an alternative widget order ("the children of $s1$ are either *fromOrder* OR *toOrder*").

After we have detected a layout pattern, we use the API provided by ORC Layout [30, 31] to adjust the layout specification by instantiating and adding the observed pattern. If no patterns can be matched anymore and unmatched edit operations are still remaining, then this means that larger parts of the layout structure have simply been replaced by different layouts, e.g. as shown in Figure 6. In this case we find the smallest subtree containing the respective changes and specify the two alternatives as logical disjunction ("either *subTree1* OR *subTree2*". This is a sign of uncommon or drastic changes in the UI, as discussed below.

For example, in Figure 1, the difference between the first two layouts is that the "Font" button is replaced by its expanded version. According to the above rules, the edit operation *replaceNode* indicates that this is an alternative layout pattern. The "Font" button and its expanded version are alternatives. The difference between the third and the forth layouts is that the "Styles Pane" button is added. As it is a widget (leaf node), the edit operation *addNode* is mapped to an optional widget pattern.

7.2 Visualizing Reconstructed Layout Quality

To detect potential resize issues in layouts, designers often need a manual process to inspect the huge space of all potential device and layout dimensions to verify that there are no problems in the layouts. To address this challenge, we propose an error map that uses colors to help the designer pinpoint various interface dimensions that may be in need of improvement and/or repair. The map enables designers to see a visual overview of specific points in the resize space to enable them to quickly target and repair potential areas of concern.

We visualize the quality of the reconstructed layout in an *error map* using three metrics: structural error, transition error, and "fault lines". The size of the error map matches the (scaled) size of the maximum layout, and sampled layout sizes correspond to points in the map. To define the structural error of a layout at a certain size, we consider the corresponding tabstops of the original layout and the corresponding reconstructed layout. The structural error of a layout is the sum of the squared differences between the positions of corresponding tabstops, divided by the number of tabstops. As illustrated in Figures 1 and 7, the color of the error map at each point corresponds to the structural error of the sampled layout, with darker shades of yellow indicating larger error and linear color gradients filled in between the sampled points.

As shown in Figure 7, beyond structural error, the error map also visualizes the transition error (green / blue), which measures the pixel difference between the sizes of the original and the reconstructed layout at which a certain transition takes place (*e.g.*, a widget moving onto a new row): green parts indicate that the original GUI transitions at a larger size than the reconstructed GUI, and blue parts indicate that the original GUI transitions at a smaller size. For example, in the error map in Figure 7, the vertical blue area shows the transition error between (Max) and (4). The left boundary of the area is the transition position in the original layout and the right one is the transition position in the reconstructed layout. This area indicates that widget 3 disappears at a (slightly) smaller width in the original layout than the reconstructed layout. The vertical green area demonstrates the transition error between (4) and (1) indicating that widget 2 reflows to the next row at a (slightly) larger width in the original layout compared to the reconstructed layout. Analogously, the horizontal blue area shows that

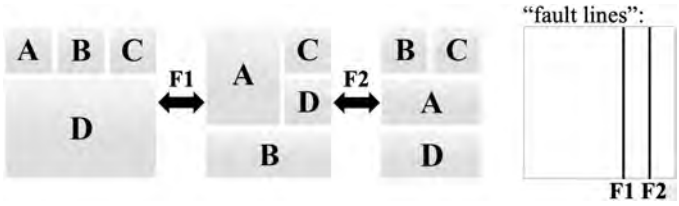

Figure 6: Visualization of "fault lines" in the error map of a layout with 'bad' (i.e. potentially confusing) behaviors.

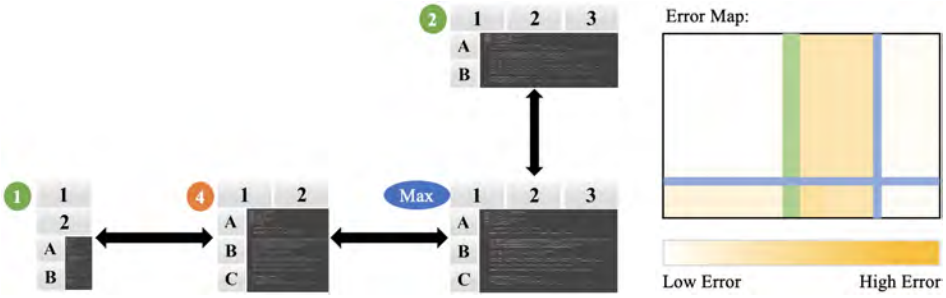

Figure 7: Visualization of reconstruction quality for the example in Figure 3 with an *error map*.

widget B disappears at a (slightly) smaller height in the original layout between (Max) and (2).

To highlight layout transitions that are potentially confusing to the user, *i.e.*, where widgets switch positions in surprising ways, we also identify such "fault" lines. In Figure 6 we show such a problematic layout, where the fault lines F1 and F2, which correspond to the transitions on the left, are shown as black lines in the error map on the right. More specifically, we show fault lines when a) widgets are reordered or b) larger parts of the layout structure change, as indicated by layout alternatives that cannot be matched to common ORC Layout patterns, *i.e.*, where an OR needs to be inserted between two whole sublayout alternatives. Fault lines indicate transition positions that might need adjustment in the reverse engineered specification. In Figure 6, we show a layout with resize behaviors that cannot be predicted with common ORC Patterns. Both transition positions have fault lines, which illustrate the points where an unpredictable behavior occurs and where the layout could potentially be improved. Thus, designers could use fault lines as guides to identify and fix bad layout behaviours, *e.g.*, by modifying the generated ORC layout specification. For example, this layout could be changed to a horizontal flow layout to exhibit better resize behavior.

8 APPLICATIONS

It is often time-consuming for designers to create new resizable UIs from scratch. Sometimes a designer might find a UI that is similar to what they are looking for. ReverseORC can help designers to reconstruct ORC Layout specifications for existing UIs and then use those specifications in other applications and on other platforms. In the following, we briefly discuss this for the MS Word "ribbon" GUI and the BBC News website, which both use highly dynamic layouts. Furthermore, we briefly discuss how ReverseORC can help designers to modify, extend and even create resizable GUIs by example.

8.1 GUI Reverse Engineering – MS Word Ribbon

ReverseORC can be used on dynamic, hand-coded GUI layouts, such as the well-known MS Word "ribbon" toolbar. In Figure 1, we present our reverse engineering result for the "ribbon". The yellow lines in the original UI samples on the left illustrate the layout structure results of each sample. The edit operations detected for the transitions between them are shown with blue arrows. On the right, the corresponding reconstructed UI with its ORC Layout patterns are shown, exhibiting the same layout behaviours as the original.

8.2 Web UI Reverse Engineering – BBC News

ReverseORC enables moving layouts across platforms. Designers may want to replicate a web layout in a mobile app or vice versa. As our system is platform and framework independent, this means that a layout can be re-used in another form of applications, as we can unify different layouts by reverse engineering. For example, we can reverse engineer GUI layouts for the web, and web layouts for GUIs. As a demonstration of this, Figure 8 shows the reverse engineering result for the BBC News website into a GUI environment, which opens up options for cross-platform applications. Our method works well for webpages that have well-defined and reasonably predictable layout methods, but we cannot claim that our method works well for all layout methods that exist on the web. Consider for example a tiled layout that rearranges tiles randomly upon a resize. In this case, ReverseORC creates a very large layout specification that contains many OR clauses.

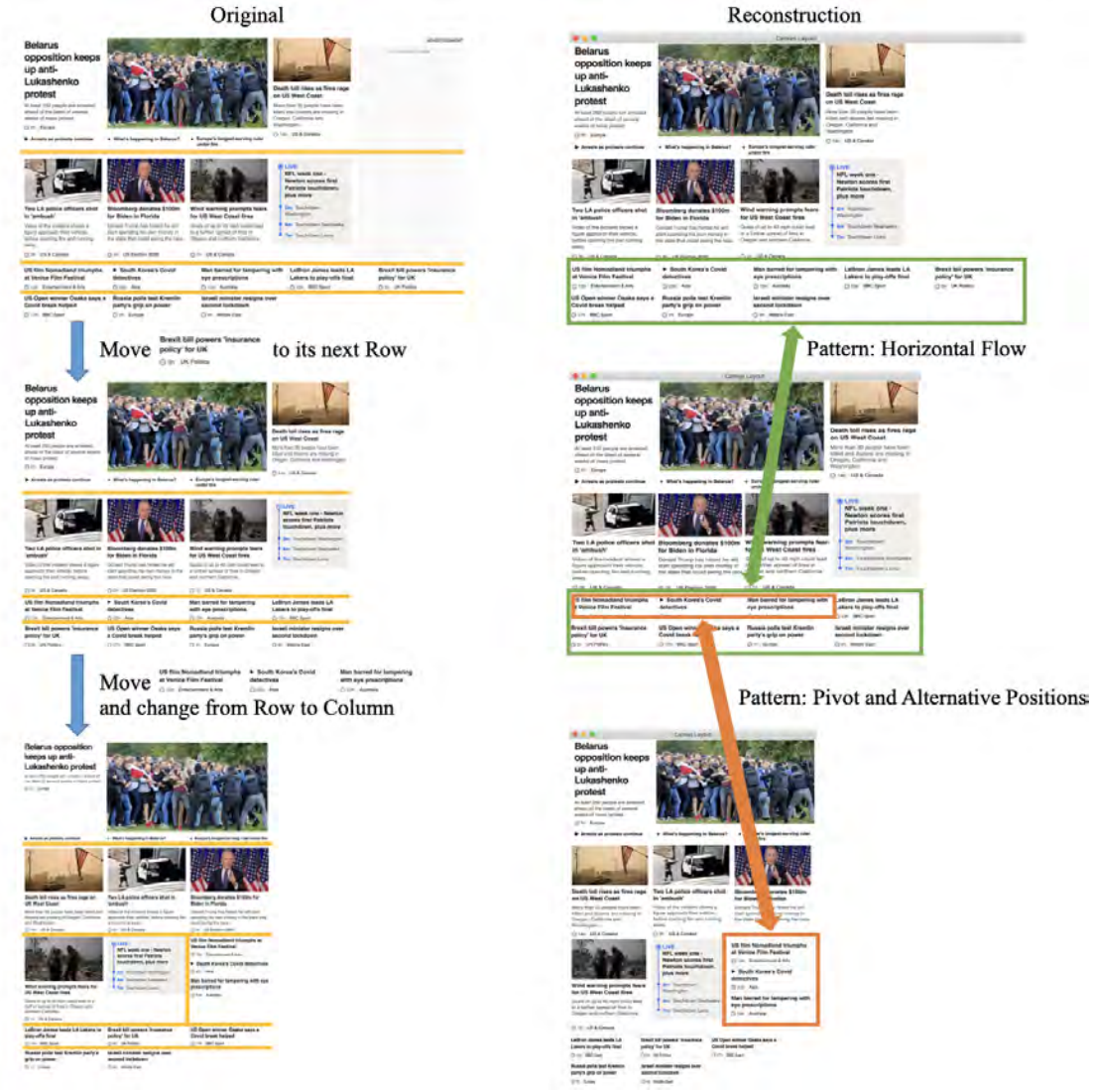

Figure 8: Reverse engineering result for the BBC News website, displayed as an ORC Layout GUI. One Horizontal Flow pattern was omitted in the figure for space reasons.

8.3 Exemplar-Based Layout Design

The layout difference detection and ORC layout pattern matching parts of ReverseORC can be used to reverse engineer desired layout behaviors and generate resizable GUIs based on examples, *e.g.*, multiple different sizes of a static GUI layout drawn by a visual designer. Given multiple such static GUI layouts drawn by a visual designer for different window sizes, ReverseORC detects layout differences among the static exemplar layouts and infers an ORC layout that matches the drawn layout results and the designer's intention (Figure 9).

If the results do not match the designer's expectations, they can iteratively draw new exemplars, or change their existing exemplars, and ReverseORC's difference detection will pick up the differences and change the layout specification accordingly. This can be used, for example, to disambiguate some layout behaviors by providing more examples, or add extra transitions for a smoother resize behaviour. The designer could pick a respective size by clicking on the error map, and then modify or replace the UI for the chosen size in a UI builder-like interface.

Similarly, manual exemplars can be combined with exemplars that are sampled automatically. For example, in Figure 1, the transition at the fault line between layout four and five rearranges two sublayouts in a fairly arbitrary manner to use available width. Upon seeing the fault line, a designer could manually re-draw layout five, e.g. in a manner that re-arranges the widgets according to a flow layout. ReverseORC would then create a Flow pattern for the transition and the fault line would disappear.

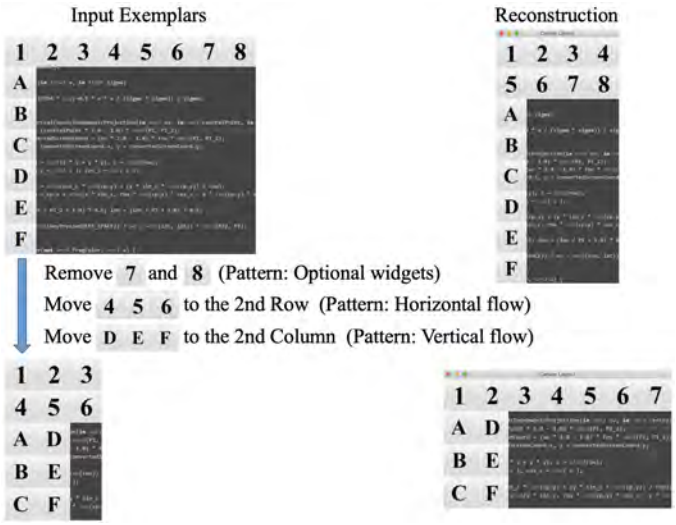

Figure 9: Given multiple different sizes of a static GUI layout drawn by a visual designer, ReverseORC can generate an ORC layout specification based on the patterns inferred from the drawn layout results. The two figures on the right show the results of resizing the GUI reconstructed from the left two window sizes.

9 DISCUSSION

ReverseORC is an implementation- and platform-independent reverse engineering approach to detect layout behaviors and types and to generate a matching high-level ORC Layout specification for a given layout. It enables the creation of responsive, flexible layouts for existing and new applications. ReverseORC is an efficient tool; it took only about 0.4 seconds to reverse engineer the MS word "Ribbon" toolbar and about 0.5 second for the BBC News website on an average laptop computer. It can be used as part of the design process for making existing UIs more flexible, to fix problems in existing GUIs, and in developing completely new GUIs. In effect, we can deal with most layout methods, including flow, grid, grid-bag, and ORC layouts. One caveat is that we currently cannot reconstruct some of the numerical parameters influencing a layout, such as the weights in grid-bag layouts and their ability to center content. This is a topic for future work.

Developers interact with ReverseORC by editing UI exemplars (Section 8.3). Such editing of a UI's appearance has been well studied [60, 73] and found to be developer-friendly, especially when compared to specifying interactive behaviors directly [46]. Similar to Expresso [33], ReverseORC developers also specify UI exemplars simply by dragging and resizing elements, and this was already demonstrated to be easy and useful by Expresso. The usefulness of the resulting ORC specifications has been validated in [30, 31].

A drawback of using accessibility APIs to extract widget information is that in some applications, not all the widgets might provide accessibility APIs. Hybrid techniques combining pixel-based methods and accessibility API could further improve the accuracy of widget detection [27]. For websites, sampling is predominantly a one-dimensional problem as window widths are much more important. Due to the affordance of vertical scrolling, heights are relatively less relevant. Thus, instead of a binary grid search, a simpler approach for website layouts might be to sample different widths through a one-dimensional binary interval search.

If the designer or the implementer of a layout manager made a severe mistake when a GUI was designed, which causes unexpected widget placement to occur in a layout, e.g., an optional widget that "flickers in and out" during resizing, then our framework will typically create many alternative patterns – since our ReverseORC approach can only detect known layout types and patterns. While this is a fundamental limitation of our approach, it is not an algorithmic one, as we are in this case not dealing with well-defined layout behaviors. On other other hand, our approach could also be used as a sanity test for layouts to detect bugs and/or unexpected behaviors, as fault lines would appear in the error map for many such behaviors. The user can then use the results of our algorithm to replace unexpected behaviors in the layout with more deterministic and predictable patterns.

10 CONCLUSION AND FUTURE WORK

We presented ReverseORC, a novel layout reverse engineering method that reconstructs layout specifications for existing UIs, considering not only the static structure of the original but also its dynamic resize behaviors. By sampling layout sizes with a binary grid search, ReverseORC detects topological differences between layouts of different sizes, further infers layout behaviors, and generates a corresponding ORC Layout specification to enable layout customization and generation of new UIs. To our knowledge, ReverseORC is the first approach for reverse engineering dynamic resizable layouts and generating a platform independent high-level layout specification for them. We envision that our method could be widely applied in various applications and platforms. ReverseORC is available as open source from https://github.com/YueJiang-nj/ReverseORC-CHI2021.

REFERENCES

[1] Robert St Amant, Mark O Riedl, Frank E Ritter, and Andrew Reifers. 2005. Image Processing in Cognitive Models with SegMan. In *Proceedings of the 11th International Conference on Human-Computer Interaction, HCII*, Vol. 2005.

[2] Pavol Bielik, Marc Fischer, and Martin Vechev. 2018. Robust Relational Layout Synthesis from Examples for Android. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 156 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276526

[3] Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. 2005. Automation and Customization of Rendered Web Pages. In *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology* (Seattle, WA, USA) *(UIST '05)*. Association for Computing Machinery, New York, NY, USA, 163–172. https://doi.org/10.1145/1095034.1095062

[4] Alan Borning, Richard Kuang-Hsu Lin, and Kim Marriott. 2000. Constraint-based document layout for the Web. *Multimedia systems* 8, 3 (2000), 177–189.

[5] Deng Cai, Shipeng Yu, Ji-Rong Wen, and Wei-Ying Ma. 2003. *VIPS: a Vision-based Page Segmentation Algorithm.* Technical Report MSR-TR-2003-79. 28 pages. https://www.microsoft.com/en-us/research/publication/vips-a-vision-based-page-segmentation-algorithm/

[6] Tsung-Hsiang Chang, Tom Yeh, and Robert C. Miller. 2010. GUI Testing Using Computer Vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, Georgia, USA) *(CHI '10)*. Association for Computing Machinery, New York, NY, USA, 1535–1544. https://doi.org/10.1145/1753326.1753555

[7] Yu Chen, Xing Xie, Wei-Ying Ma, and Hong-Jiang Zhang. 2005. Adapting web pages for small-screen devices. *IEEE internet computing* 9, 1 (2005), 50–56.

[8] Morgan Dixon and James Fogarty. 2010. Prefab: Implementing Advanced Behaviors Using Pixel-Based Reverse Engineering of Interface Structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, Georgia, USA) *(CHI '10)*. Association for Computing Machinery, New York, NY, USA, 1525–1534. https://doi.org/10.1145/1753326.1753554

[9] Morgan Dixon, James Fogarty, and Jacob Wobbrock. 2012. A General-Purpose Target-Aware Pointing Enhancement Using Pixel-Level Analysis of Graphical Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Austin, Texas, USA) *(CHI '12)*. Association for Computing Machinery, New York, NY, USA, 3167–3176. https://doi.org/10.1145/2207676.2208734

[10] Morgan Dixon, Gierad Laput, and James Fogarty. 2014. Pixel-Based Methods for Widget State and Style in a Runtime Implementation of Sliding Widgets. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Toronto, Ontario, Canada) *(CHI '14)*. Association for Computing Machinery, New York, NY, USA, 2231–2240. https://doi.org/10.1145/2556288.2556979

[11] Morgan Dixon, Daniel Leventhal, and James Fogarty. 2011. Content and Hierarchy in Pixel-Based Methods for Reverse Engineering Interface Structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vancouver, BC, Canada) *(CHI '11)*. Association for Computing Machinery, New York, NY, USA, 969–978. https://doi.org/10.1145/1978942.1979086

[12] Morgan Dixon, Alexander Nied, and James Fogarty. 2014. Prefab Layers and Prefab Annotations: Extensible Pixel-Based Interpretation of Graphical Interfaces. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology* (Honolulu, Hawaii, USA) *(UIST '14)*. Association for Computing Machinery, New York, NY, USA, 221–230. https://doi.org/10.1145/2642918.2647412

[13] Carmel Domshlak, Samir Genaim, and Ronen Brafman. 2000. Preference-based configuration of web page content. In *14th European Conference on Artificial Intelligence (ECAI 2000), Configuration Workshop, Berlin, Germany*. 19–22.

[14] W. Keith Edwards, Scott E. Hudson, Joshua Marinacci, Roy Rodenstein, Thomas Rodriguez, and Ian Smith. 1997. Systematic Output Modification in a 2D User Interface Toolkit. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology* (Banff, Alberta, Canada) *(UIST '97)*. Association for Computing Machinery, New York, NY, USA, 151–158. https://doi.org/10.1145/263407.263537

[15] Jan P Finis, Martin Raiber, Nikolaus Augsten, Robert Brunel, Alfons Kemper, and Franz Färber. 2013. Rws-diff: flexible and efficient change detection in hierarchical data. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. 339–348.

[16] Jun Fujima, Aran Lunzer, Kasper Hornbæk, and Yuzuru Tanaka. 2004. Clip, Connect, Clone: Combining Application Elements to Build Custom Interfaces for Information Access. In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology* (Santa Fe, NM, USA) *(UIST '04)*. Association for Computing Machinery, New York, NY, USA, 175–184. https://doi.org/10.1145/1029632.1029664

[17] John Gerdes. 2009. User Interface Migration of Microsoft Windows Applications. *Journal of Software Maintenance* 21 (05 2009), 171–187. https://doi.org/10.1002/smr.400

[18] Saul Greenberg and Bill Buxton. 2008. Usability Evaluation Considered Harmful (Some of the Time). In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Florence, Italy) *(CHI '08)*. Association for Computing Machinery, New York, NY, USA, 111–120. https://doi.org/10.1145/1357054.1357074

[19] Björn Hartmann, Leslie Wu, Kevin Collins, and Scott R. Klemmer. 2007. Programming by a Sample: Rapidly Creating Web Applications with d.Mix. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology* (Newport, Rhode Island, USA) *(UIST '07)*. Association for Computing Machinery, New York, NY, USA, 241–250. https://doi.org/10.1145/1294211.1294254

[20] Masatomo Hashimoto and Akira Mori. 2008. Diff/TS: A tool for fine-grained structural change analysis. In *2008 15th working conference on reverse engineering*. IEEE, 279–288.

[21] Osamu Hashimoto and Brad A. Myers. 1992. Graphical Styles for Building Interfaces by Demonstration. In *Proceedings of the 5th Annual ACM Symposium on User Interface Software and Technology* (Monteray, California, USA) *(UIST '92)*. Association for Computing Machinery, New York, NY, USA, 117–124. https://doi.org/10.1145/142621.142635

[22] Hiroshi Hosobe. 2005. Solving Linear and One-Way Constraints for Web Document Layout. In *Proceedings of the 2005 ACM Symposium on Applied Computing* (Santa Fe, New Mexico) *(SAC '05)*. Association for Computing Machinery, New York, NY, USA, 1252–1253. https://doi.org/10.1145/1066677.1066959

[23] Scott E. Hudson, Jennifer Mankoff, and Ian Smith. 2005. Extensible Input Handling in the SubArctic Toolkit. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Portland, Oregon, USA) *(CHI '05)*. Association for Computing Machinery, New York, NY, USA, 381–390. https://doi.org/10.1145/1054972.1055025

[24] Scott E. Hudson and Shamim P. Mohamed. 1990. Interactive Specification of Flexible User Interface Displays. *ACM Trans. Inf. Syst.* 8, 3 (July 1990), 269–288. https://doi.org/10.1145/98188.98201

[25] Scott E. Hudson and Ian Smith. 1997. Supporting Dynamic Downloadable Appearances in an Extensible User Interface Toolkit. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology* (Banff, Alberta, Canada) *(UIST '97)*. Association for Computing Machinery, New York, NY, USA, 159–168. https://doi.org/10.1145/263407.263539

[26] Scott E. Hudson and Kenichiro Tanaka. 2000. Providing Visually Rich Resizable Images for User Interface Components. In *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology* (San Diego, California, USA) *(UIST '00)*. Association for Computing Machinery, New York, NY, USA, 227–235. https://doi.org/10.1145/354401.354783

[27] Amy Hurst, Scott E. Hudson, and Jennifer Mankoff. 2010. Automatically Identifying Targets Users Interact with during Real World Tasks. In *Proceedings of the 15th International Conference on Intelligent User Interfaces* (Hong Kong, China) *(IUI '10)*. Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/1719970.1719973

[28] Dugald Ralph Hutchings and John Stasko. 2005. Mudibo: Multiple Dialog Boxes for Multiple Monitors. In *CHI '05 Extended Abstracts on Human Factors in Computing Systems* (Portland, OR, USA) *(CHI EA '05)*. Association for Computing Machinery, New York, NY, USA, 1471–1474. https://doi.org/10.1145/1056808.1056944

[29] Jaekyu Ha, R. M. Haralick, and I. T. Phillips. 1995. Recursive X-Y cut using bounding boxes of connected components. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, Vol. 2. 952–955 vol.2. https://doi.org/10.1109/ICDAR.1995.602059

[30] Yue Jiang, Ruofei Du, Christof Lutteroth, and Wolfgang Stuerzlinger. 2019. ORC Layout: Adaptive GUI Layout with OR-Constraints. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, Article 413, 12 pages. https://doi.org/10.1145/3290605.3300643

[31] Yue Jiang, Wolfgang Stuerzlinger, Matthias Zwicker, and Christof Lutteroth. 2020. ORCSolver: An Efficient Solver for Adaptive GUI Layout with OR-Constraints. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3313831.3376610

[32] Solange Karsenty, James A. Landay, and Chris Weikart. 1993. Inferring Graphical Constraints With Rockit. In *Proceedings of the Conference on People and Computers VII* (York, United Kingdom) *(HCI'92)*. Cambridge University Press, 137–153. https://doi.org/10.1007/3-540-58601-_91

[33] R. Krosnick, S. W. Lee, W. S. Laseck, and S. Onev. 2018. Expresso: Building Responsive Interfaces with Keyframes. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 39–47. https://doi.org/10.1109/VLHCC.2018.8506516

[34] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: Automating & Sharing How-to Knowledge in the Enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Florence, Italy) *(CHI '08)*. Association for Computing Machinery, New York, NY, USA, 1719–1728. https://doi.org/10.1145/1357054.1357323

[35] James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A. Lau. 2009. End-User Programming of Mashups with Vegemite. In *Proceedings of the 14th International Conference on Intelligent User Interfaces* (Sanibel Island, Florida, USA) *(IUI '09)*. Association for Computing Machinery, New York, NY, USA, 97–106. https://doi.org/10.1145/1502650.1502667

[36] Greg Little, Tessa A. Lau, Allen Cypher, James Lin, Eben M. Haber, and Eser Kandogan. 2007. Koala: Capture, Share, Automate, Personalize Business Processes on

the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '07)*. Association for Computing Machinery, New York, NY, USA, 943–946. https://doi.org/10.1145/1240624.1240767

[37] Christof Lutteroth. 2008. Automated Reverse Engineering of Hard-Coded GUI Layouts. In *Proceedings of the Ninth Conference on Australasian User Interface - Volume 76* (Wollongong, Australia) *(AUIC '08)*. Australian Computer Society, Inc., AUS, 65–73. https://doi.org/10.5555/1378337.1378350

[38] Christof Lutteroth, Robert Strandh, and Gerald Weber. 2008. Domain specific high-level constraints for user interface layout. *Constraints* 13, 3 (2008), 307–342.

[39] Christof Lutteroth and Gerald Weber. 2006. User Interface Layout with Ordinal and Linear Constraints. In *Proceedings of the 7th Australasian User Interface Conference - Volume 50* (Hobart, Australia) *(AUIC '06)*. Australian Computer Society, Inc., AUS, 53–60. https://doi.org/10.5555/1151758.1151764

[40] Ethan Marcotte. 2011. *Responsive Web Design*. A book apart.

[41] Melody Moore and Spencer Rugaber. 1997. Using Knowledge Representation to Understand Interactive Systems. In *Proceedings of the 5th International Workshop on Program Comprehension (WPC '97) (WPC '97)*. IEEE Computer Society, USA, 60.

[42] Melody M. Moore. 1996. Rule-Based Detection for Reverse Engineering User Interfaces. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96) (WCRE '96)*. IEEE Computer Society, USA, 42.

[43] Melody Marie Moore, James D. Foley, and Spencer Rugaber. 1998. *User Interface Reengineering*. Ph.D. Dissertation. USA. AAI9918460.

[44] Melody M. Moore and Lilia Moshkina. 2000. Migrating Legacy User Interfaces to the Internet: Shifting Dialogue Initiative. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00) (WCRE '00)*. IEEE Computer Society, USA, 52.

[45] Brad Myers, Scott E. Hudson, and Randy Pausch. 2000. Past, Present, and Future of User Interface Software Tools. *ACM Trans. Comput.-Hum. Interact.* 7, 1 (March 2000), 3–28. https://doi.org/10.1145/344949.344959

[46] Brad Myers, Sun Young Park, Yoko Nakano, Greg Mueller, and Amy Ko. 2008. How designers design and program interactive behaviors. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE, 177–184.

[47] Brad A. Myers. 1995. User Interface Software Tools. *ACM Trans. Comput.-Hum. Interact.* 2, 1 (March 1995), 64–103. https://doi.org/10.1145/200968.200971

[48] Brad A. Myers, Richard G. Mcdaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Ieee Computer Society, Ieee Computer Society, Andrew Mickish, Alex Klimovitski, and Patrick Doane. 1997. The Amulet Environment: New Models for Effective User Interface Software Development. *IEEE Transactions on Software Engineering* 23 (1997), 347–365.

[49] George Nagy and Sharad C Seth. 1984. Hierarchical representation of optically scanned documents. (1984).

[50] Tuan Anh Nguyen and Christoph Csallner. 2015. Reverse Engineering Mobile Application User Interfaces with REMAUI. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, 248–259. https://doi.org/10.1109/ASE.2015.32

[51] Jeffrey Nichols and Tessa Lau. 2008. Mobilization by Demonstration: Using Traces to Re-Author Existing Web Sites. In *Proceedings of the 13th International Conference on Intelligent User Interfaces* (Gran Canaria, Spain) *(IUI '08)*. Association for Computing Machinery, New York, NY, USA, 149–158. https://doi.org/10.1145/1378773.1378793

[52] Dan R. Olsen. 2007. Evaluating User Interface Systems Research. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology* (Newport, Rhode Island, USA) *(UIST '07)*. Association for Computing Machinery, New York, NY, USA, 251–258. https://doi.org/10.1145/1294211.1294256

[53] Dan R. Olsen, Scott E. Hudson, Thom Verratti, Jeremy M. Heiner, and Matt Phelps. 1999. Implementing Interface Attachments Based on Surface Representations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Pittsburgh, Pennsylvania, USA) *(CHI '99)*. Association for Computing Machinery, New York, NY, USA, 191–198. https://doi.org/10.1145/302979.303038

[54] Dan R. Olsen, Trent Taufer, and Jerry Alan Fails. 2004. ScreenCrayons: Annotating Anything. In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology* (Santa Fe, NM, USA) *(UIST '04)*. Association for Computing Machinery, New York, NY, USA, 165–174. https://doi.org/10.1145/1029632.1029663

[55] Richard L Potter. 1992. *Triggers: Guiding Automation with Pixels to Achieve Data Access*. University of Maryland, Center for Automation Research, Human/Computer Interaction Laboratory.

[56] Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, and Jesús García Molina. 2014. Model-driven reverse engineering of legacy graphical user interfaces. *Automated Software Engineering* 21, 2 (2014), 147–186.

[57] Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, Jesús García Molina, and Jean Vanderdonckt. 2016. A layout inference algorithm for Graphical User Interfaces. *Information and Software Technology* 70 (2016), 155–175.

[58] Alireza Sahami Shirazi, Niels Henze, Albrecht Schmidt, Robin Goldberg, Benjamin Schmidt, and Hansjörg Schmauder. 2013. Insights Into Layout Patterns of Mobile User Interfaces by an Automatic Analysis of Android Apps. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems* (London, United Kingdom) *(EICS '13)*. ACM, Gothenburg, Sweden, 275–284. https://doi.org/10.1145/3197231.3197249

[59] Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, and Jesús García Molina. 2010. Model-Driven Reverse Engineering of Legacy Graphical User Interfaces. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (Antwerp, Belgium) *(ASE '10)*. Association for Computing Machinery, New York, NY, USA, 147–150. https://doi.org/10.1145/1858996.1859023

[60] Adriano Scoditti and Wolfgang Stuerzlinger. 2009. A New Layout Method for Graphical User Interfaces. In *Science and Technology for Humanity (TIC-STH), 2009 IEEE Toronto International Conference*. IEEE, 642–647. https://doi.org/10.1016/j.infsof.2015.10.005

[61] Robert St. Amant, Henry Lieberman, Richard Potter, and Luke Zettlemoyer. 2000. Programming by Example: Visual Generalization in Programming by Example. *Commun. ACM* 43, 3 (March 2000), 107–114. https://doi.org/10.1145/330534.330549

[62] Stefan Staiger. 2007. Static Analysis of Programs with Graphical User Interface. In *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*. 252–264. https://doi.org/10.1109/CSMR.2007.44

[63] E. Stroulia, M. El-Ramly, P. Iglinski, and P. Sorenson. 2003. User Interface Reverse Engineering in Support of Interface Migration to the Web. *Automated Software Engg.* 10, 3 (July 2003), 271–301. https://doi.org/10.1023/A:1024460315173

[64] Wolfgang Stuerzlinger, Olivier Chapuis, Dusty Phillips, and Nicolas Roussel. 2006. User Interface Façades: Towards Fully Adaptable User Interfaces. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology* (Montreux, Switzerland) *(UIST '06)*. Association for Computing Machinery, New York, NY, USA, 309–318. https://doi.org/10.1145/1166253.1166301

[65] Amanda Swearngin, Amy J. Ko, and James Fogarty. 2017. Genie: Input Retargeting on the Web through Command Reverse Engineering. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 4703–4714. https://doi.org/10.1145/3025453.3025506

[66] Desney S. Tan, Brian Meyers, and Mary Czerwinski. 2004. WinCuts: Manipulating Arbitrary Window Regions for More Effective Use of Screen Space. In *CHI '04 Extended Abstracts on Human Factors in Computing Systems* (Vienna, Austria) *(CHI EA '04)*. Association for Computing Machinery, New York, NY, USA, 1525–1528. https://doi.org/10.1145/985921.986106

[67] J. Vanderdonckt, L. Bouillon, and N. Souchon. 2001. Flexible Reverse Engineering of Web Pages with VAQUISTA. In *Proceedings Eighth Working Conference on Reverse Engineering*. 241–248. https://doi.org/10.1109/WCRE.2001.957828

[68] Yuan Wang, David J DeWitt, and J-Y Cai. 2003. X-Diff: An effective change detection algorithm for XML documents. In *Proceedings 19th international conference on data engineering (Cat. No. 03CH37405)*. IEEE, 519–530.

[69] Gerald Weber. 2010. A Reduction of Grid-Bag Layout to Auckland Layout. In *Proceedings of the 2010 21st Australian Software Engineering Conference (ASWEC '10)*. IEEE Computer Society, 67–74. https://doi.org/10.1109/ASWEC.2010.38

[70] Xing Xie, Chong Wang, Li-Qun Chen, and Wei-Ying Ma. 2005. An adaptive web page layout structure for small devices. *Multimedia Systems* 11, 1 (2005), 34–44.

[71] Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. 2009. Sikuli: Using GUI Screenshots for Search and Automation. In *Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology* (Victoria, BC, Canada) *(UIST '09)*. Association for Computing Machinery, New York, NY, USA, 183–192. https://doi.org/10.1145/1622176.1622213

[72] Brad Vander Zanden and Brad A. Myers. 1990. Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Seattle, Washington, USA) *(CHI '90)*. ACM, Seattle, Washington, USA, 27–34. https://doi.org/10.1007/978-3-319-67744-_2

[73] Clemens Zeidler, Christof Lutteroth, Gerald Weber, and Wolfgang Stürzlinger. 2012. The Auckland Layout Editor: An Improved GUI Layout Specification Process. In *Proceedings of the 13th International Conference of the NZ Chapter of the ACM's Special Interest Group on Human-Computer Interaction* (Dunedin, New Zealand) *(CHINZ '12)*. Association for Computing Machinery, New York, NY, USA, 103. https://doi.org/10.1145/2379256.2379287

[74] C. Zeidler, G. Weber, A. Gavryushkin, and Christof Lutteroth. 2017. Tiling algebra for constraint-based layout editing. *J. Log. Algebraic Methods Program.* 89 (2017), 67–94.

[75] Luke S. Zettlemoyer and Robert St. Amant. 1999. A Visual Medium for Programmatic Control of Interactive Applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Pittsburgh, Pennsylvania, USA) *(CHI '99)*. Association for Computing Machinery, New York, NY, USA, 199–206. https://doi.org/10.1145/302979.303039

[76] Luke S. Zettlemoyer, Robert St. Amant, and Martin S. Dulberg. 1998. IBOTS: Agent Control through the User Interface. In *Proceedings of the 4th International Conference on Intelligent User Interfaces* (Los Angeles, California, USA) *(IUI '99)*. Association for Computing Machinery, New York, NY, USA, 31–37. https://doi.org/10.1145/291080.291087

Algorithm 1: Tabstop Creation

```
1  Function TabstopCreation(L)
2      xtabs : (ℤ → X-Tabstop) = {0 ↦ L.leftTab, L.width ↦
         L.rightTab}
3      ytabs : (ℤ → Y-Tabstop) = {0 ↦ L.topTab, L.height ↦
         L.bottomTab}
4      for w ∈ L.Widgets do
5          if w.left ∈ Domain(xtabs) then
6              w.leftTab ← xtabs(w.left)
7          end
8          else
9              xtabs ← xtabs ∪ {w.left ↦ w.leftTab}
10         end
11         Process right, top, bottom tabstops analogously.
12     end
13     return xtabs, ytabs
14 end
```

Algorithm 2: Tabstop Layout Dividers

```
1  Function GetTabstopLayoutDividers(tabs, widgets, xy)
2      tabValues ← sorted(tabs.keys())[1 : −1]
3      tabstopLayoutDividers ← []
4      for tabValue ∈ tabValues do
5          divideLayout ← True
6          for w ∈ widgets do
7              if xy == "x" then
8                  minBoundary ← w.left
9                  maxBoundary ← w.right
10             end
11             if xy == "y" then
12                 minBoundary ← w.top
13                 maxBoundary ← w.bottom
14             end
15             if minBoundary < tabValue and
                 maxBoundary > tabValue then
16                 divideLayout ← False
17             end
18         end
19         if divideLayout == True then
20             tabstopLayoutDividers ←
                 tabstopLayoutDividers ∪ {tabValue}
21         end
22     end
23     return tabstopLayoutDividers
24 end
```

A TABSTOP CREATION

For each layout L, we define tabstops through two functions $xtabs()$ and $ytabs()$ that map from positions in the GUI to tabstop variables in the layout (Algorithm 1). We define $xtabs$ as a function mapping from x-coordinates to x-tabstops. Initially, it contains two elements that map the leftmost x-position in the GUI to the left tabstop variable of the layout and correspondingly for the right (Line 2). $ytabs$ is the analogous mapping for y-coordinates to y-tabstops (Line 3).

Line 4-12 show how we loop through all the widgets in the GUI to create mappings to the two tabstop functions $xtabs$ and $ytabs$. We check if the x-coordinate of the current widget's left boundary $w.left$ is contained in the domain of xtabs, *i.e.*, this x-coordinate $w.left$ is already mapped to an existing x-tabstop in the function $xtabs$ (Line 5). In practice, there might be some small displacements in the layout. For example, a widget might be displaced by a pixel due to a rounding error. Then it is unreasonable to add two tabstops with a one-pixel distance in between. Thus, instead of checking whether $w.left$ is in the domain of the function $xtabs$, we can check whether there exists an x-coordinate $xpos$ in $xtabs$ that is within a tolerance value ϵ. If so, we map $w.leftTab$ to $xtabs(w.left)$ to eliminate near-duplicate tabstop variables and near-identical mappings in the function $xtabs$ (Line 6). If $w.left$ could not be mapped to a tabstop variable in the function $xtabs$, then we insert a new mapping from the x-coordinate $w.left$ to the tabstop variable $w.leftTab$ (Line 9). We process all three other boundaries of each widget (right, top, bottom) analogously.

In the end, the algorithm yields the final $xtabs$ and $ytabs$ functions. Also, we now have four unique tabstop variables for each widget in the layout.

B TABSTOP LAYOUT DIVIDERS

We define a tabstop as a tabstop layout divider if it is a clean cut dividing the layout into two parts where the tabstop does not cross any widget in the layout. For a horizontal tabstop layout divider, all the widgets in the layout are either above it or below it, analogously for vertical tabstop layout dividers. We first get all the tabstops that divide the layout into two parts (Line 2). Then we loop over each of these tabstops, we check if all the widgets in the layout have minimum boundary greater than the tabstop value or maximum boundary less than the tabstop. (For the x-axis, the minimum boundary of a widget is its left boundary and the maximum its right boundary, while for y-axis, they are the top and bottom boundaries respectively.) If so, the tabstop is a clean cut for the layout, and thus a tabstop layout divider (Line 4-22).

C LAYOUT STRUCTURE RECONSTRUCTION

We reconstruct the layout structure by recursively subdividing it based on layout dividers (Algorithm 3). As the basic case in this recursion, if the current sublayout only contains a single widget, we simply return its identity (Line 1). Otherwise, we first try to subdivide the layout using vertical layout dividers (Line 3-4). If such subdivision is possible (Line 6), then we sort all the widgets in the layout by their bottom boundary positions (Line 8). We then assign the widgets to different sublayouts based on the positions of the horizontal layout dividers, and recursively use the reconstruction on each sublayout structure (Line 9-19). We merge two layout dividers into one if there is no widgets between them (Line 11).

We aim to reconstruct the simplest possible layout structure. Therefore, to avoid creating layout dividers caused by accidental

Algorithm 3: Layout Structure Construction

```
1  Function ConstructLayoutStructure(L)
2      if layout is a single widget, return it
3      xtab, ytab ← TabstopCreation(L)
4      ytabLayoutDividers ←
         GetTabstopLayoutDividers(ytab, L.widgets, "y")
5      layoutStructure ← {"Column" : []}
6      if ytabLayoutDividers not empty then
7          widgetList ← []
8          widgetsCurr ← L.widgets sorted by w.bottom
9          for tabValue ∈ ytabLayoutDividers do
10             sublayoutWidgets ← {w |w ∈
                 widgetsCurr ∧ w.bottom ≤ tabValue}
11             if sublayoutWidgets is empty, then continue
12             layoutStructure["Column"] ←
                 layoutStructure["Column"] ∪
                 {ConstructLayoutStructure(Layout(
                 sublayoutWidgets))}
13             widgetList ← widgetList ∪ {sublayoutWidgets}
14             widgetsCurr ←
                 widgetsCurr − sublayoutWidgets
15         end
16         if widgetsCurr not empty then
17             layoutStructure["Column"] ←
                 layoutStructure["Column"] ∪
                 {ConstructLayoutStructure(Layout(
                 sublayoutWidgets))}
18             widgetList ← widgetList ∪ {widgetsCurr}
19         end
20         layoutStructure["Column"][i : j] ←
             ConstructLayoutStructure(Layout(
             merge(widgetList[i : j]))) if possible simplified
             structure exists for any
             0 ≤ i < j < len(layoutStructure["Column"])
21         return layoutStructure
22     end
23     process xtab analogously
24     if no subdivision is possible, then return L
25 end
```

alignments, we regroup widgets in multiple consecutive sublayouts and try running the algorithm recursively to simplify the resulting layout structure. We reconstruct the sublayout if we can get a simplified sublayout structure by grouping them (Line 20). We try horizontal subdivision (with vertical layout dividers) first as it is more common and in line with reading order. If horizontal subdivision is not possible, we process vertical subdivision analogously (Line 23). If both cases are impossible, which is very rare as UIs are typically laid out using a division-based containment hierarchy, then the layout can only be described using tabstops directly (Line 24), *e.g.*, in a pinwheel layout [74]. Figure 4 shows the visualization of the constructed layout structure of the MS Word "ribbon".

Algorithm 4: Identical Node Detection

```
1  Function DetectIdenticalNode(S1Curr, S2Curr)
2      for s2 ∈ S2Curr do
3          if s2.hashCode ∈ S1HashMap.keys() then
4              S1Curr ← S1Curr − s1 if s1 ∈ S1
5              S2Curr ← S2Curr − s2
6              if s1 ∉ S1 then
7                  changes += {moveNode(s1, s2)}
8              end
9              else
10                 pairs ← pairs ∪ {s2 ↦ s1}
11             end
12         end
13     end
14 end
```

D NODE PROPERTIES

Based on the resulting layout tree structure of an input layout specification gotten from Algorithm 3, we traverse this tree structure and create a corresponding new tree. Each widget becomes a leaf node, while each *Row* or *Column* structure becomes an internal node (non-leaf node). Each node stores the following properties:

- *widgetId / type*:
  - leaf nodes: widget identifier
  - internal nodes: structure type (either "*Row*" or "*Column*")
- *parent*: parent node of the current node
- *children*: the list of children nodes of the current node
- *pathToRoot*: a list of tuples containing the ancestors of the current node along with the positions among their siblings, (*e.g.,* If *pathToRoot* of the current node is $[(Root, 2), (A, 3), (B, 4)]$, then A is the 2nd child of the *Root* node, B is the 3rd child of A, and the current node is the 4th child of B.)
- *leaves*: the list of all the leaves in the subtree rooted at the current node
- *hashCode*:
  - leaf nodes: $hash(widgetId)$ based on a standard hash function
  - internal nodes: $hash$("*Row*"/"*Column*") $+ 1 \times child1.hashCode + 2 \times child2.hashCode + 3 \times child3.hashCode + \ldots$, where $child1, child2, child3, etc.$ are children nodes of the current node
- *childHashCode*:
  - leaf nodes: same as *hashCode*
  - internal nodes: *child*1.*hashCode XOR child*2.*hashCode XOR child*3.*hashCode XOR* ..., where *child*1, *child*2, *etc.* are children nodes of the current node

E IDENTIFYING CORRESPONDING NODES

To identify corresponding nodes in the sibling lists $S1$ and $S2$, we first loop over the *S2Curr* list to find all the nodes $s2 \in S2$ such that there is a node $s1 \in Tree1$ with the same *hashCode* as $s2$ (Algorithm 4). If the corresponding node $s1$ does not belong to $S1$, then $s1$ moved to $S2$ at the position it occurs in $S2$ (Line 6-8).

Algorithm 5: Similar Node Detection

```
1  Function DetectSimilarNodes(S1Curr, S2Curr)
2      for s2 ∈ S2Curr do
3          if s2.childHashCode ∈ S1ChildHashMap.keys()
             then
4              S1Curr ← S1Curr − s1 if s1 ∈ S1
5              S2Curr ← S2Curr − s2
6              if s1 ∉ S1 then
7                  changes ← changes ∪ {moveNode(s1, s2)}
8              end
9              else
10                 pairs ← pairs ∪ {s2 ↦ s1}
11             end
12             if s1.type ≠ s2.type then
13                 changes ← changes ∪
                     {changeType(s1, s1.type, s2.type)}
14             end
15             if s1.children ≠ s2.children then
16                 changes ← changes ∪
                     {changeChildrenOrder(s1, s1.children, s2.children)}
17             end
18         end
19     end
20 end
```

Otherwise, we pair $a1$ and $s2$ in sibling lists (Line 10). We expect most $s2 \in S2$ have its corresponding node $s1 \in Tree1$. Thus, after this step, *S2Curr* list should only contain very few nodes. Similarly, for all the remaining nodes $s2 \in S2Curr$, we check whether there is a node $s1 \in Tree1$ with same *childHashCode* as $s2$, and pair them accordingly (Algorithm 5 Line 2-11). In addition, we identify type changes and children node order changes (Line 12-17).

F LAYOUT DIFFERENCE DETECTION

We detect the differences between two layout specifications and identify edit operations by recursively comparing corresponding lists of sibling nodes $S1$ and $S2$ (Algorithm 6), with the corresponding lists containing child nodes of nodes that have already been determined to correspond. The basic idea is: we try to match corresponding nodes in S2 with nodes in Tree1. Whenever we have found a correspondence, we compare the respective nodes and record edit operations for any differences in position, type or child order. Initially, the inputs of the layout difference detection algorithm are $S1$ = [*root node of Tree1*] and $S2$ = [*root node of Tree2*], where *Tree1* and *Tree2* are the trees representing the two layout specifications. We then identify the corresponding nodes in the sibling lists and recursively apply this algorithm to the children nodes of corresponding nodes. As we find corresponding nodes, we specify their differences (if any) as edit operations and add them to a set *changes*. In each call, we maintain a hash table *pairs* mapping nodes in $S2$ to corresponding nodes in $S1$; with *pairs* initially empty (Line 2). We keep track of currently unpaired nodes by removing the paired nodes from the lists $S1$ and $S2$ once a pair has been found.

Algorithm 6: Layout Difference Detection

```
1  Function LayoutDifferenceDetection(S1, S2)
2      pairs ← {} // hash table from nodes in S2 to nodes in S1
3      S1Curr ← S1
4      S2Curr ← S2
5      DetectIdenticalNodes(S1Curr, S2Curr) // find and pair S2
         nodes that are identical to Tree1 nodes
6      DetectSimilarNodes(S1Curr, S2Curr) // find and pair S2
         nodes that have the same leaves as Tree1 nodes
7      while S2Curr not empty do
8          (s1Best, s2Best) ← (s1 ∈ S1, s2 ∈ S2) s.t. max
             number of common leaves
9          maxSim ← numCommonLeaves(s1Best, s2Best)
10         if maxSim > 0 then
11             S1Curr ← S1Curr − s1
12             S2Curr ← S2Curr − s2
13             pairs ← pairs ∪ {s2Best ↦ s1Best}
14             if s1Best.type ≠ s2Best.type then
15                 changes ← changes ∪
                     {changeType(s1Best, s2Best.type)}
16             end
17             LayoutDifferenceDetection(s1Best.children,
                 s2Best.children)
18         end
19         else
20             break
21         end
22     end
23     S1Paired ← [s1 for s1 ∈ S1 if s1 ∈ pairs.values()]
24     S2Paired ← [s2 for s2 ∈ S2 if s2 ∈ pairs.keys()]
25     Replace s2 ∈ S2Paired by pairs[s2]
26     if S1Paired ≠ S2Paired then
27         changes ← changes ∪
             {changeChildrenOrder(s1.parent, S2Paired)}
28     end
29     if num of paired nodes before s1 ∈ S1 = num of paired
         nodes before s2 ∈ S2 then
30         S1Curr ← S1Curr − s1
31         S2Curr ← S2Curr − s2
32         changes ← changes ∪ {replaceNode(s1, s2)}
33     end
34     for s1 ∈ S1Curr do
35         changes ← changes ∪ {removeNode(s1)}
36     end
37     for s2 ∈ S2Curr do
38         changes ← changes ∪ {addNode(s2)}
39     end
40 end
```

We first try to detect strong correspondences based on a node's hash values in *IdenticalNodeDetection* and *DetectSimilarNodes*. These methods loop over *S2Curr* to find all the nodes $s2 \in S2$ such that there is a node $s1 \in Tree1$ with the same *hashCode* or

childHashCode as $s2$. If the corresponding node $s1$ does not belong to $S1$, then $s1$ moved to $S2$ at the position it occurs in $S2$. Otherwise, we pair $a1$ and $s2$ in sibling lists. In addition, we identify type changes and also children node order changes (Details see Appendix Section E). We expect most $s2 \in S2$ have its corresponding node $s1 \in Tree1$. Thus, after this step, $S2Curr$ list should only contain very few nodes.

If we cannot find corresponding nodes for all the nodes $s2 \in S2Curr$, we pair the remaining nodes in $S1Curr$ and $S2Curr$ based on their similarity. We keep pairing ($s1 \in S1Curr, s2 \in S2Curr$) depending on the largest number of common leaves and recursively call the algorithm on their children nodes. We stop this pairing process when there is no node remaining in $S2Curr$ or all the $s1 \in S1Curr$ and $s2 \in S2Curr$ have no common leaves (Algorithm 6 Line 7-22). In addition, we check whether the order of all the paired nodes has not changed in $S1$ and $S2$ (Line 23-28).

After all the above pairing operations, $S1Curr$ and $S2Curr$ contain all the nodes that cannot be paired with any node in the other layout tree. If $s1 \in S1Curr$ and $s2 \in S2Curr$ have the same relative position among their sibling nodes, *i.e.,* the number of paired nodes before them are the same, then we infer that $s1$ in $S1$ is replaced by $s2$ in $S2$ (Line 29-33). All the remaining $s1 \in S1Curr$ are removed from $S1$ and $s2 \in S2Curr$ are added in $S2$ (Line 34-39).