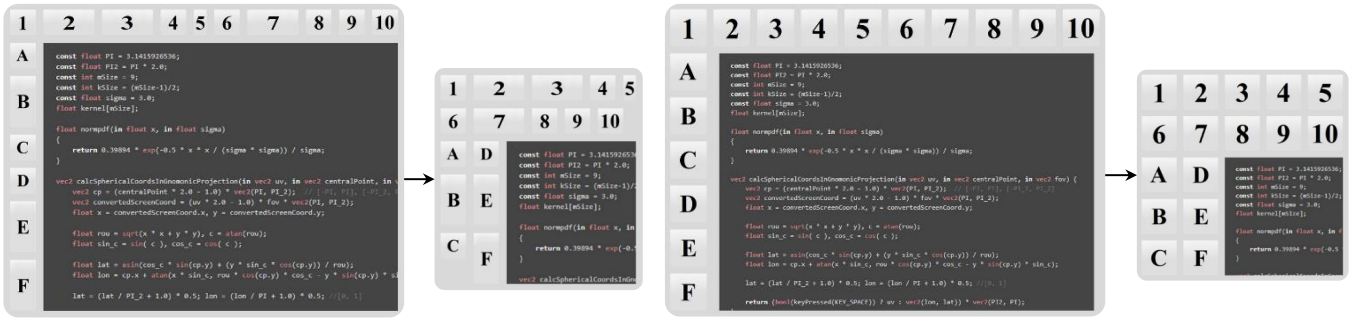
ORC Layout: Adaptive GUI Layout with OR-Constraints

Yue Jiang
Department of Computer Science
University of Maryland, College Park, MD, USA
yuejiang@cs.umd.edu

Ruofei Du
Google LLC, San Francisco, CA &
University of Maryland, College Park, MD, USA
me@duruofei.com

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

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



(a) two flow layouts nested in a grid (b) ORC Layout

Figure 1: Examples of GUIs combining elements of grid and flow layouts, which can only be realized with ORC layouts. Both (a) and (b) show the results before and after resizing the window. (a) shows a traditional solution with two flow layouts nested in a grid. It looks irregular since the original sizes of the widgets in the toolbars are not the same. Using ORC layouts in (b), we can add constraints that cross-cut the GUI hierarchy, ensuring that the tool buttons all have the same size and hence consistent appearance.

ABSTRACT

We propose a novel approach for constraint-based graphical user interface (GUI) layout based on OR-constraints (ORC) in standard soft/hard linear constraint systems. ORC layout unifies grid layout and flow layout, supporting both their features as well as cases where grid and flow layouts individually fail. We describe ORC design patterns that enable designers to safely create flexible layouts that work across different screen sizes and orientations. We also present the ORC Editor, a GUI editor that enables designers to apply ORC in a safe and effective manner, mixing grid, flow and new ORC layout features as appropriate. We demonstrate that our prototype can adapt layouts to screens with different aspect ratios with only a single layout specification, easing the burden of GUI maintenance. Finally, we show that ORC specifications can be modified interactively and solved efficiently at runtime.

CCS CONCEPTS

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

KEYWORDS

GUI builder, layout manager, constraint-based layout, visual interface design, visual programming

ACM Reference Format:
Yue Jiang, Ruofei Du, Christof Lutteroth, and Wolfgang Stuerzlinger. 2019. ORC Layout: Adaptive GUI Layout with OR-Constraints. In *CHI Conference on Human Factors in Computing Systems Proceedings*

CHI 2019, May 4–9, 2019, Glasgow, Scotland Uk

ACM ISBN 978-1-4503-5970-2/19/05…$15.00
https://doi.org/10.1145/3290605.3300643

(CHI 2019), May 4–9, 2019, Glasgow, Scotland Uk. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3290605.3300643

1 INTRODUCTION

Automatically computed layouts are widely used in many applications, including webpages, graphical user interfaces (GUIs), documents, slides, and mobile apps. WYSIWYG ("what you see is what you get") GUI editors enable designers to create layouts by defining the positions and sizes of widgets, as well as specifying how a layout adapts to changes, through layout models. Grid layout models such as grid-bag can adjust the relative size of widgets, while keeping mutual alignments intact. Flow layout models can break widgets into new rows or columns, but are limited in the way widgets can be aligned. Constraint-based layout models, the most powerful option, are becoming widespread and can be used to specify any grid layout, can align widgets across different groups, and relax constraints between aligned widgets to enable more flexibility. For example, Apple's AutoLayout [28] supports adaptive interfaces on devices ranging from desktops to smartphones. CSS's Flex(ible) Box[1], widely supported in web browsers, applies constraints for dynamically fitting the content and solving alignment problems.

Despite their power, current constraint-based layout models have limitations. First, they do not support flow layouts. Second, for GUIs to work well in both portrait and landscape orientations and on small as well as large screens, separate layout specifications have to be created for each case and activated on demand. Current constraint-based layout models are not flexible enough to deal with all such changes directly, which increases the workload of the designer: several layout specifications have to be created and maintained. During the evolution of a GUI, designers then have to manually synchronize changes between different layout specifications. If all cases could be covered in a single layout specification, this could make GUI design more efficient and less error-prone.

Specifying flexible layouts with constraints is not easy, and specifications are typically formulated as a linear programming problem, through a system of linear equations and inequalities, which are either hard or soft constraints. A dedicated solver then calculates the position and size of each widget. A specification can be ambiguous and lead to unpredictable results, or be conflicting and have no result at all. Such problems may be "hidden" in a specification and only show in some cases, *e.g.*, for a particularly small screen.

In this paper, we propose ORC layouts – a novel approach to constraint-based layout that supports OR-constraints in a standard soft/hard linear constraint system. An OR-constraint is a disjunction of multiple constraints (only one part needs to be true to satisfy the OR-constraint) where the whole OR-constraint is a hard constraint and each part is a soft one. Our method allows designers to specify a layout that can adapt to drastically different screen sizes and orientations using a single layout specification. A GUI can then be adapted to changes at runtime without having to consider several separate specifications, avoiding inefficiencies and inconsistencies. ORC layout also unifies the – so far separate – grid and flow layout models, addressing both of their shortcomings. Additionally, it offers some entirely new features that are currently unsupported by other layout models, such as optional widgets.

ORC layouts can be constructed interactively with our ORC Editor. It enables designers to instantiate templates for the ORC layout patterns presented here, and adjust them by modifying their parameters. The ORC editor then automatically maintains all low-level constraints, so designers do not need to deal with them directly.

Figure 1 (b) shows an example that combines elements of grid and flow layouts, only achievable through ORC layouts. On a large screen, the GUI has a toolbar at the top and one on the left, with an edit pane taking up most of the space. For a small screen, the toolbar at the top can break into two rows and the left one into two columns, using a horizontal and a vertical flow layout. While similar behaviour can be achieved with two flow layouts nested in a grid, the results would look different if the original sizes of the toolbar widgets are not the same (Figure 1 (a)). With OR-constraints we can cross-cut the GUI hierarchy, ensuring that the tool buttons all have the same size and hence consistent appearance. Many other layout functionalities can be realized with OR-constraints, including balanced flow layouts and layout alternatives.

Contributions

We contribute two new ideas for constraint-based GUI layout:

(1) We present ORC layouts, a new powerful way to express GUI layouts, which seamlessly integrate OR-constraints into standard constraint systems for layout specifications. Various layout patterns are supported. One of the biggest benefits is that OR-constraints can unify flow layouts and grid layouts, which opens up options for layouts that work across screen rotations and cases where grid and flow layouts individually fail.
(2) We present a new approach to solving constraint systems with soft/hard linear and OR-constraints at interactive rates. When resizing a canvas at runtime, our algorithm solves the constraint system based on the previous results. Thus, we can solve a system with 175 constraints in less than a second for operations such as insert, delete, move, and resize.

[1] CSS Flexible Box: https://w3schools.com/css/css3_flexbox.asp

2 RELATED WORK

Our research draws from several related ideas in GUI layout, such as systems of soft/hard linear constraints, minimum/preferred/maximum constraints, CSS, layout solvers, and GUI builders, which we outline in the following.

Layout Models

Layout models are widely used to specify the layout of user interfaces in a way that is (to a certain degree) size independent. Layout managers then determine at runtime the actual GUI layout based on the constraints and the context, such as window size. See also an overview of early approaches [22, 23]. Simple layout models, such as group, table, grid, and grid-bag layouts, align widget groups horizontally or vertically and often nest layout containers hierarchically. While intuitive, these models often implicitly add more constraints than desired, which introduces maintenance issues [19, 41]. They cannot align widgets across a hierarchy, a problem that constraints can address [20]. Similar to text layout, flow layouts break widgets into new rows or columns. Object-oriented models such as Amulet [25] allow developers to combine some properties of flow and grid layouts programmatically.

More modern layout models use constraints, through varying constraint types and explicit or implicit constraint specification. Linear constraint systems [2, 3, 5, 13, 18] are powerful and used today fairly widely, *e.g.*, in Apple's Auto Layout [28]. They use sets of equalities and inequalities to specify absolute and relative alignment of widgets as well as absolute and relative widget sizes. Unlike simpler layout models, linear constraint systems have a well-understood mathematical basis, which makes results less dependent on implementation details. Linear constraint systems can be combined, allowing reuse of existing specifications in a modular manner [20]. Most layout models can be reduced to linear constraint systems [34, 42], with the exception of flow layouts. Our work addresses this shortcoming.

Except for flow layouts, existing layout models support only static widget topologies: widget cannot change their position relative to widgets they are aligned with. Flow layouts enable such repositioning in a limited manner: widgets can overflow into a new row or column. Our ORC layouts permit designers to specify flexible layouts that mix grid and flow layouts and seamlessly integrate multiple layout alternatives.

Soft/Hard Linear Constraint Systems

Soft/hard linear constraint systems can differentiate between crucial layout characteristics (*e.g.*, that a widget must have a certain size to be usable) and merely desirable ones (*e.g.*, that two buttons should have the same size). Two types of constraints express this: hard constraints, which must be satisfied, and soft ones, which can be neglected when it is impossible to determine a solution that can satisfy all constraints in the system. Further, not all soft constraints should be treated equally, as some soft constraints are more important than others [4]. Hence, a weight is used to rank each soft constraint, with higher weights being equivalent to higher priorities. Hard constraints can be considered as soft ones with infinite weight.

The intrinsic sizes of each widget are typically defined by min/max/preferred size constraints. The minimum usually depends on the contents of the widget and the maximum is normally limited by the window or screen size, while the preferred size is pre-defined, often by the designer of the GUI editor, or customized by the UI designer in the editor.

Layout Solvers

Various solvers exist for linear constraint-based GUI specifications [2, 3, 5, 13, 38]. Many of them use linear or quadratic programming [2, 3, 5, 18], *i.e.*, use a linear or quadratic objective function to take constraint priorities into account. Such objective functions penalize deviations from desired positions and sizes, which can improve aesthetics upon resizing [40]. Yet, these solvers will try to satisfy *all* constraints simultaneously, subject to priorities in case of conflicts. But they cannot express alternatives, as they support only conjunctions of constraints (AND), but do not permit disjunctions (OR).

Z3 [6] is a very powerful satisfiability modulo theory (SMT) solver. It supports linear constraints with priorities, other types of constraints, and first-order logic, and is thus able to solve specifications with disjunctive constraints. Using incremental solving, it can perform better if the current constraint system is similar to the previous one. Z3 has previously been suggested for solving formalized CSS layout specifications [27] and to support layout editing with direct manipulation of individual constraints [14]. Yet, Z3 has not been used for general GUI/widget layout with OR-constraints before.

GUI Builders/Editors

GUI builder tools support the interactive specification of widgets and layouts and directly generate the corresponding layout models and constraints. GUI builders need to enable designers to directly and quickly specify complex, flexible layouts, while preventing layout errors. Their set of editing operations determine the range of layouts and resize behaviours that can be specified by a designer. Core challenges include: 1) powerful layout models support constraints which can be difficult to visualize intuitively, and 2) layout specifications may contain errors that manifest only for particular GUI sizes.

FormsVBT [1] used both textual and visual layout representations, with parallel editing. Gilt [12] introduced reusable styles and visual tabs to simplify layout and appearance specification. Opus [15] supported visual manipulation of simple layout constraints. Gleicher [11] proposed a differential editing approach, where constrained graphical objects can be manipulated directly and changes are continuously reflected in all objects. Unidraw [33] showed how direct manipulation can be implemented by simulating widgets.

Specifying individual layout constraints can be cumbersome. Thus, some have proposed to infer simple constraints "by demonstration". Rockit [17] infers constraints on 2D graphical objects based on their locations. Peridot [24], Druid [31], and Lapidary [37] generate code based on user interface drawings and example interactions.

Recent GUI builders have focused on creating appropriate layout constraints quickly and robustly through direct manipulation. The Intui builder [30] allows designers to quickly specify the resize behaviour of a GUI by aligning parts of the GUI and designating them either as rigid "struts" or flexible "springs". The Auckland Layout Editor [39] enables designers to specify a constraint-based layout, while automatically keeping the specification solvable and free of overlap. This work also used multiple previews to directly visualize how a layout looks at distinct window sizes. GUI programming by manipulation [14] also permits editing of individual layout constraints. Designers can interactively disable and enable individual constraints to resolve resulting ambiguities.

None of the above GUI builders allow designers to specify layouts with OR-constraints.

Layout Alternatives and Alternative Generators

Linear constraint layouts provide some flexibility for resizing a GUI. However, if a GUI needs to be rendered on a large variety of devices, then a common solution is for the designers to define several layout alternatives, and to select dynamically among them, depending on the rendering context, (*e.g.*, available screen size, resolution, and aspect ratio). This approach is common for mobile apps [29, 36] and "responsive web design" [21]. For mobile apps, the class of device is used to choose among separate layout specifications, while for responsive web design CSS media queries can be specified to apply different layout alternatives, *e.g.*, based on screen size. While flexible, the CSS framework does not offer the same capabilities as ORC layouts, without constructing layouts programmatically with JavaScript. For example, CSS layouts often use a "fluid grid", *i.e.*, a rigid grid-like layout in which elements can flow over into separate rows. Yet, one cannot direct particular widgets to flow in a specific way, as in the connected layout pattern (Figure 5). Unlike the ORC approach, a GUI designer still has to manually specify several exact layout alternatives as well as the rules that determine when each alternative should be used.

To automatically improve the user experience based on device properties, tasks and user requirements, some work proposed GUI adaptation, *i.e.*, the automatic modification of GUIs based on specific criteria [35]. Fogarty et al. [7] used cost functions to generate improved GUIs. SUPPLE [8, 9] uses optimization to generate adapted user interfaces, *e.g.*, by replacing widgets or grouping them differently in containers, and also to personalize interfaces for users with disabilities [10]. Automatic layout generation tools generate proposed alternatives for a given layout [32, 36]. Then, designers only need to choose and modify the generated suggestions. A complementary approach is adaptive layout templates [16], which also can generate layout alternatives. Overall, designers are often limited to the (few) alternatives that such tools provide.

In contrast to previous work, our work aims to enable designers to directly specify flexible GUI layouts. With ORC layouts, designers can leave it up to the solver to determine when to apply a (or a combination of) layout alternative(s), based on screen space or other desirable criteria. Designers can also prioritize, combine, and conditionally use alternatives. For example, a good ORC layout for a smallish medium-sized screen might combine alternatives for medium and small screens, *e.g.*, compress only the toolbar but not other UI parts. ORC layouts also can synchronize layout changes with constraints, *e.g.*, to ensure that several rows/columns are broken simultaneously and in a similar manner to achieve a balanced appearance (Figure 6), even if they are not next to each other.

3 OR-CONSTRAINED (ORC) LAYOUTS

Here we describe how OR-constrained (ORC) layouts are defined, edited, and solved.

ORC Layout Specifications

In ORC layouts, and as in other constraint-based layouts, the min/max/preferred size of each widget and the relationships among widgets are all defined by constraints. In addition to hard and soft constraints, the constraint system for an ORC layout also contains OR-constraints. OR-constraints enable the designer to specify multiple alternative constraints, only one of which needs to be met, *e.g.*, a toolbar that is on the left side in landscape but is repositioned to the top (or bottom) of the screen in portrait mode. The designer only needs to specify both (or any number of) alternative constraints joined with an OR-clause. Thus, with OR-constraints, a single layout specification can support many different screen sizes and aspect ratios.

We define OR-constraints through a mixture of hard and soft constraints. A whole ORC clause has a hard constraint,

while each part of the OR-constraint has an independent soft constraint. This ensures layout specifications are always feasible. Some individual soft-constrained parts of an OR-constraint can be given higher priority than others through weights, which defines the level of importance or priority of that part. For example, in the OR-constraint "to the right OR at the beginning of the next row", we likely prefer to place the current widget to the right of the previous one. Thus, we attach a larger, *e.g.*, double, weight to "to the right" part.

ORC Editor

The ORC Editor looks and works similar to other constraint-based GUI builders, with a canvas, palette, and properties panel [39]. It supports safe use of ORC layout patterns (see Section 4) through parameterised templates. Layouts can be edited interactively in the canvas through direct manipulation, or using the properties panel, *e.g.*, for template instantiation and modification. Designers can declaratively choose templates for widgets and widget containers and specify their parameters; the editor automatically creates and maintains the corresponding constraints. The template-based approach simplifies GUI maintenance compared to programming approaches [22], and a single flexible specification instead of several alternatives can ease authoring by reducing redundancy. As in ALE [39], users can still add and edit individual constraints, but we expect this to be used infrequently. Similar to ALE, the editor automatically solves the constraint system after each change and keeps the layout solvable by detecting, highlighting, and disabling conflicting constraints.

Solving a System with OR-Constraints

The naïve approach to solve a system with OR-constraints is to try all solutions. For very small systems, this might be feasible but does not scale well: for every OR-constraints with two parts, there are two potential solutions. This number grows exponentially, *e.g.*, with only ten OR-constraints there are 1024 potential solutions that need to be evaluated. The naïve strategy quickly becomes impractical.

OR-constrained systems are a subset of SMT (Satisfiability Modulo Theory) problems. Instead of an explicit solver algorithm, we thus use Z3 [6], a very efficient SMT solver. Z3 typically scored highly in the yearly SMT competitions[2] during the last decade. One of the key insights of our work is that SMT solvers can be used to solve GUI layouts with OR-constraints. Z3 has the ability to solve specifications with disjunctive sets of constraint alternatives, which makes it an ideal tool to solve a system of constraints that includes OR-constraints. Z3 chooses constraints by considering all possible combinations of alternatives and evaluating their appropriateness for the given situation with an objective function. The solver tries to satisfy as many constraints as possible, based on their assigned priorities. We have not experienced Z3 convergence failures. How constraints interact with each other can be controlled in two ways: 1) by setting constraint priorities, *e.g.*, prefer A over alternative A2, or 2) by combining constraints in different ways using Boolean operators, *e.g.*, (A AND B) OR (A2 AND B2) will force the solver to pick alternative branches in the same way.

[2]SMT competitions: smtcomp.org

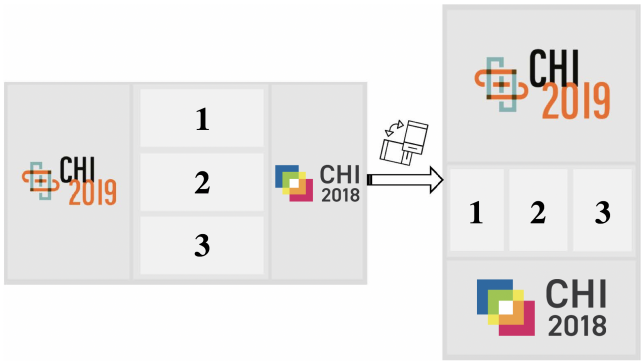


Figure 2: An ORC layout with a single layout specification for horizontal and vertical rotation. Classic layouts, such as flow and grid, can either not specify the change in alignment between the widgets depending on the window aspect ratio, or cannot ensure consistent sizes across the different widget groups.

Re-solving the entire constraint system iteratively after each change of a layout can be time-consuming and may, *e.g.*, slow down the speed of interactive GUI editing. To produce layout results dynamically at runtime, computations need to be efficient. By reusing parts of the previous solution, Z3 supports incremental solving, *i.e.*, it produces a result faster if the current constraint system is similar to the previous one. Instead of re-creating the whole constraint system after each adjustment in the layout, the ORC Editor removes obsolete constraints from the previously solved system and adds new ones. This re-solves only part of the new constraint system based on the previous results and achieves better performance.

ORC Layouts for Screen Rotation

OR-constraints can unify flow layouts and grid layouts, which opens up options for layouts to work across screen rotations and cases where grid and flow layouts individually fail. This enables designers to seamlessly deal with screen rotations with a single layout specification. For instance, neither grid nor flow layouts can deal with the case in Figure 2 through a single layout specification. Although the general change in topology may be achieved using two nested flow layouts, the flow layouts cannot ensure that the widgets are consistently

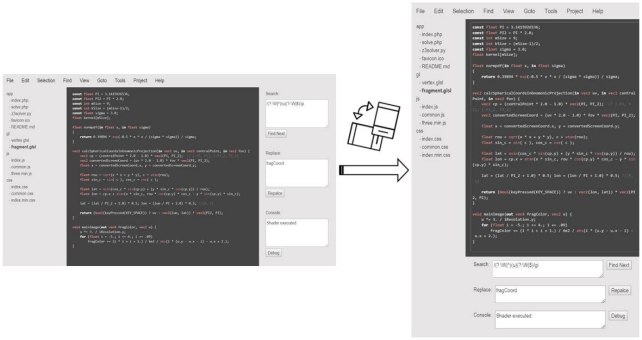


Figure 3: A practical use case of ORC layouts for dealing with different device orientations. Note that the whole layout changes from horizontal to vertical while the inner layout of the control panel changes the opposite way.

aligned as shown. There are three high-level widgets (labeled areas) in the layout: CHI 2019, block, and CHI 2018, denoted as [a], [b], and [c] respectively. Inside block [b], we have three low-level widgets: [1], [2], and [3]. Using ORC, the alignments in the two alternatives (horizontal and vertical) can be specified separately using constraints and then combined with OR. In contrast to other layout methods, ORC can also ensure consistent sizes across the two groups [a], [b], [c] and [1], [2], [3] as follows:

$$\left(\left([b]_{\text{height}} = \sum_{i=1}^{3} [i]_{\text{height}}\right) \wedge \left([b]_{\text{width}} = \max_{i=1,2,3} [i]_{\text{width}}\right)\right)$$

$$\textbf{OR}\left(\left([b]_{\text{width}} = \sum_{i=1}^{3} [i]_{\text{width}}\right) \wedge \left([b]_{\text{height}} = \max_{i=1,2,3} [i]_{\text{height}}\right)\right)$$

The two soft constraints above specify the correct alignment for the horizontal case (Figure 2 left) or the vertical case (right), respectively. A more practical use case of an ORC layout for device rotation is shown in Figure 3. This layout contains a menu, a file explorer panel, a main view for code editing, and a control panel with widgets for searching, replacing, and console debugging. It uses OR-constraints similar to the previous example.

4 ORC LAYOUT PATTERNS

ORC layout covers a superset of the most popular layout mechanisms, allowing designers to "blend" them. To achieve this in a controlled manner, we propose patterns as basic building blocks that can be combined into layouts of arbitrary complexity. For example, ORC layout patterns can specify constraints for cross-cutting (Figures 1), connecting (Figure 5), further constraining (Figures 6 and 10) and re-positioning sub-layouts (Figures 2-4 and 7), or replacing them entirely (Figure 8). Furthermore, multiple patterns can be combined (Figure 9). Here we present the constraints for several of our new layout patterns. To our knowledge no current layout mechanism or editor can deal with the shown examples, except through multiple layout specifications or code.

Mixed Layout Specifications

Previous layout managers often focus on solving a single type of layout. The use of OR-constraints makes it possible to mix different types of constraints and layout specifications. For example, a "to the right OR at the start of the next row below" constraint between each successive pair of widgets enables ORC layouts to unify flow and grid-bag layouts. With such a unified layout, the designer only needs to create a single layout specification for both landscape and portrait modes.

In the following, we assume that all widgets are ordered in the layout. We denote the i-th widget as $[i]$ and the set of all widgets as W. For each widget $[i] \in W$, we define its left & top position, width, and height as $[i]_{\text{left}}$, $[i]_{\text{top}}$, $[i]_{\text{width}}$, $[i]_{\text{height}}$ respectively. The soft constraint "to the right", C_{Right}, is formulated as:

$$C_{Right} := ([i]_{\text{left}} = [i-1]_{\text{left}} + [i-1]_{\text{width}}) \wedge \left([i]_{\text{top}} = [i-1]_{\text{top}}\right)$$

The soft constraint "at the start of the next row below", $C_{NextRow}$, is formulated as:

$$C_{NextRow} := ([i]_{\text{left}} = 0) \wedge \left([i]_{\text{top}} \geq [j]_{\text{top}} + [j]_{\text{height}}, \forall j < i, [j] \in W\right) \wedge \left(\bigvee_j [i]_{\text{top}} = [j]_{\text{top}} + [j]_{\text{height}}, \forall j < i, [j] \in W\right)$$

The OR-constraint $C_{Horizontal}$ expressing horizontal flow layout is then a hard constraint (or one with very high weight) with a disjunction of the two soft constraints "to the right" and "at the start of the next row below":

$$C_{Horizontal} := C_{Right} \;\; OR \;\; C_{NextRow}$$

Vertical flow layouts use corresponding OR-constraints for "to the bottom OR at the start of the next column". With ORC, the choice of using a horizontal or vertical flow layout can even be left to the constraint solver for flexibility by combining both alternatives with an OR, as illustrated in Figure 4.

Cross-Cutting Layout Pattern

The cross-cutting layout pattern achieves a consistent appearance in a GUI by adding constraints that cross-cut the boundaries between the different sub-layouts of a GUI. This is illustrated in Figure 1 (b), where each widget in the top toolbar has the constraint "to the right OR at the beginning of next row" and each in the left toolbar has the constraint "to the bottom OR at the beginning of next column". The toolbars automatically transform into multiple rows/columns

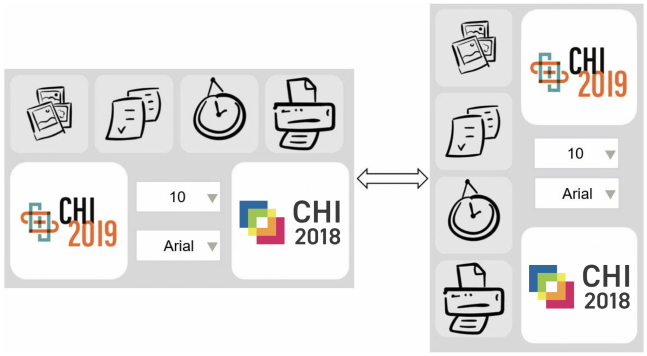


Figure 4: The horizontal window uses a "to the right OR at the start of the next row", while the vertical one has an OR-constraint "to the bottom OR at the start of the next column".

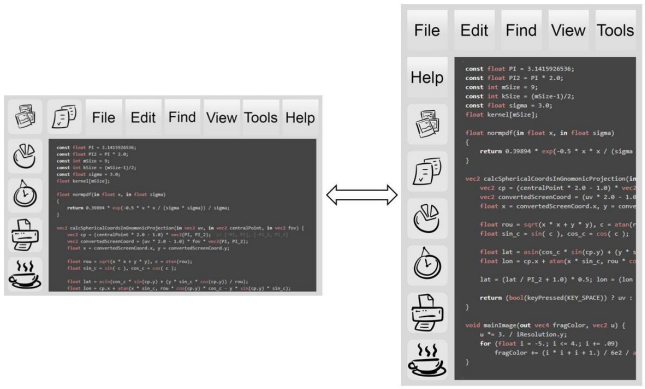


Figure 5: Connected layout pattern: the left and top toolbars are connected so widgets can move between them.

as the screen gets smaller, while cross-cutting constraints ensure that the widgets in the toolbars have sizes consistent with one another.

Connected Layout Pattern

The connected layout pattern is a generalization of different layouts, such as horizontal and vertical flow, which emerges by connecting the sub-layouts of a GUI so that widgets can move between them. This can be achieved by refining the constraints that move widgets into a new position based on available space, as discussed for flow layouts in the section about mixed layout specifications. For example, the horizontal and vertical toolbar areas in Figure 5 are connected to create a better fit in the available screen space. The most suitable number of widgets in the top toolbar is $t_{best} := window\_width \,/\, widget\_width$. If the original number of widgets in the top toolbar t is smaller than t_{best}, then the first $t_{best} - t$ widgets in the left toolbar are moved to

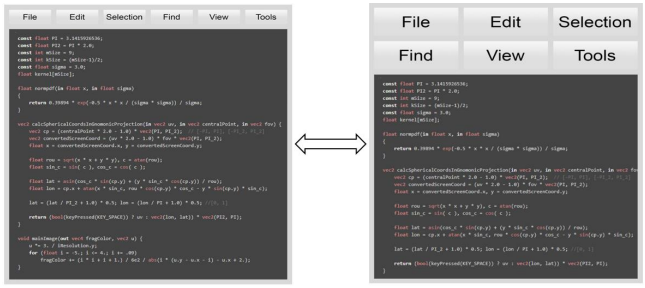


Figure 6: Balanced flow pattern: as the toolbar widgets are broken into rows, each row can only have 1, 2, 3, or 6 widgets.

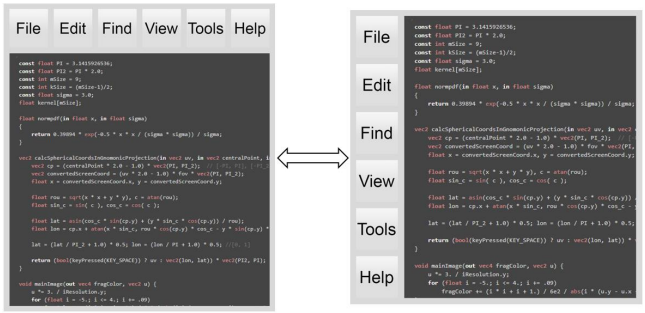


Figure 7: Alternatives positions pattern: the toolbar can be placed on top OR to the left.

the top one. Otherwise, the first $t - t_{best}$ widgets in the top toolbar are moved to the left.

Balanced Flow Pattern

Conventional flow layouts do not ensure that their rows or columns are balanced with respect to the contained widgets. ORC layouts support balanced layout alternatives and combine them with an OR. For example, when the widgets in the toolbar in Figure 6 are broken into rows, each row can only have a predefined number of widgets, which is a factor of the number of the widgets in the toolbar. We compute the set of all the factors of the number of the widgets in the toolbar. According to the window size, the top toolbar has a preferred size and number of rows. Choosing the closest number c in the factor set, t/c widgets are placed in each row.

Alternative Positions Pattern

With a simple OR-constraint, we can define alternative positions for widgets or entire sub-layouts. For example, the top toolbar in Figure 7 can be changed to be a left toolbar when the screen size changes or vice versa.

Alternative Widgets Pattern

If a window does not have enough space, some widgets can have alternative representations. For instance, *List Boxes* can be replaced by *Option Menus*. This is done by putting both of them into the layout, next to each other, but making one of them invisible by giving it a size of 0. We express this through constraints as follows, where the weight of (1) > (2):

$$(size(ListBox) = prefSize(ListBox) \wedge size(OptionMenu) = 0) \quad (1)$$
$$\text{OR } (size(ListBox) = 0 \wedge size(OptionMenu) = prefSize(OptionMenu)) \quad (2)$$

Optional Layout Pattern

When a window is made smaller, optional widgets might disappear, like in the ribbon menu in MS Word. Similar to the alternative widgets pattern, the OR-constraint "the widget has zero size OR nonzero size" realizes this functionality. For example, each widget in the ribbon menu is assigned a priority through its weight (Figure 8, Figure 9). Thus, when the window gets smaller, widgets with lower weights disappear before those with higher weights. We use the following constraints to implement this: $size = prefSize$ *OR* $size = 0$.

For widgets with high priority, $size = prefSize$ is a hard constraint (or one with a very large weight). For widgets with medium priority, $size = prefSize$ is a soft constraint with a large weight A, and $size = 0$ is a soft constraint with a small weight B. For widgets with low priority, $size = prefSize$ is a soft constraint with a large weight C, and $size = 0$ is a soft constraint with a small weight D, where weight $A >$ weight C and weight $B <$ weight D.

The example in Figure 8 shows both the widget alternative and optional functionalities. It demonstrates how the ribbon menu changes as the window gets smaller. Widgets in the ribbon menu disappear according to their weights as the window gets smaller. In the bottom row, list boxes and radio buttons are replaced by option menus to save space.

Flowing Widgets around a Fixed Area

An OR-constraint system can even deal with the case of flowing widgets around a fixed area, *e.g.*, similar to placing a picture on a page of text and flowing text around it. As shown in Figure 10 (b), with simple flow layouts a fixed widget would necessitate splitting the window into 4 parts and widgets would flow in each part separately, which results in an undesirable outcome. However, with OR-constraints, we can flow widgets around a fixed area without splitting the window into parts. The OR-constraint system can solve the whole window at the same time to arrive at a much more pleasing result and even resize widgets to avoid "ragged right" margins.

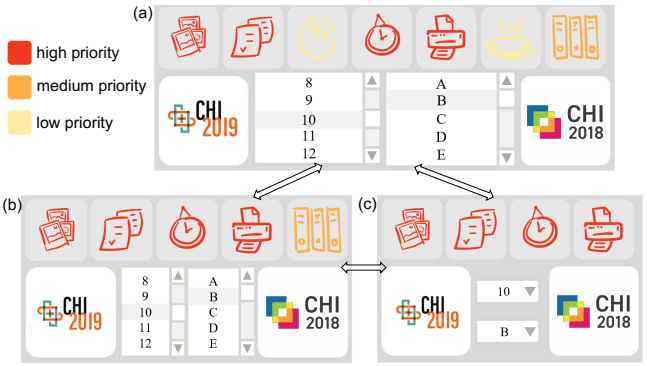


Figure 8: Optional layout and alternative widgets pattern example. (a) shows a full-sized window with all the widgets visible. (b) shows a medium-sized window where widgets with low weights disappear. (c) further reduced window size so that widgets with medium and low weights disappear. Both radio buttons and the list box are replaced by option menus to save space.

If a widget is above or below the fixed area, then it has the OR-constraint "to the right of the previous widget OR at the beginning of next row" as usual. When a row of widgets overlaps with the rows occupied by the fixed area, this will cause a row break to prevent overlap. In essence, the fixed widget cuts the rows into two parts, as in the connected layout pattern. Algorithm 1 shows how we implemented this layout. The algorithm consists entirely of conditionals and constraints with varying weights, and can therefore be translated into Boolean logic as supported by ORC. $W \rightarrow L$ is shorthand for a position constraint that places a widget W into location L. "if A then B" equals the constraint "NOT(A) OR B".

Document layout methods also address this case, but with completely different methods. For example, Adaptive Grid-Based Document Layout [16] uses adaptive layout templates to implement similar functionality, by defining sets of constraints for different cases. With ORC layout, all cases can be solved through a single constraint system.

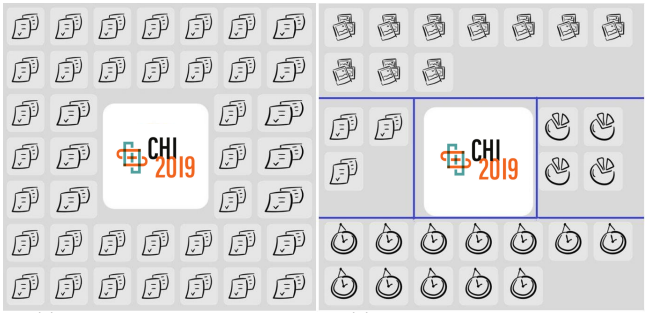
5 IMPLEMENTATION

We implemented our prototypical ORC Editor in Python 2.7 with Tkinter, a GUI library for Python, and the Z3-solver, where we used the default WMax optimization algorithm for weighted soft constraint problems [26].

6 EVALUATION

Efficiency plays a vital role in solving constraint-based systems. We measured the execution time of the ORC layout solver for different layout editing operations (insert, delete, move, and resize) with different numbers of constraints. For each condition, we measure the average execution time of 10

Figure 9: A more complex example combining the alternative position pattern for the toolbar at the top and the widgets at the bottom, the balanced flow and optional layout patterns for the toolbar, and the alternative widgets pattern for the list boxes.



(a) Flow layout *with* OR-constraints (b) Flow layout *without* OR-constraints

Figure 10: (a) Flow layout with OR-constraints. The position of the "CHI 2019" image is fixed while all the other widgets flow around it. The size of the flowing widgets adjusts to fill up the available space. (b) Flow layout without OR-constraints.

runs. We conduct the experiments on a laptop with an Intel i5 CPU. As shown in Table 1, with fewer than 175 constraints our method can efficiently solve the most common cases in less than one second on this average machine. For deployment, one could cache solved layouts for typical devices to improve performance. Note that for insert, delete, and move, our ORC Editor incrementally updates the constraint system, which saves approximately 40%-86% of the computation time for 5 widgets, and around 30%-48% of the time for 20 widgets. In contrast, a resize operation typically requires a complete rebuild of the entire constraint system, which is costlier.

Although ORC solving does not yet work at real-time interactive speeds, layouts can be solved during app loading, pre-computed in a background thread, and cached for typical screen sizes. For more complex cases, our method could also take advantage of hierarchical layouts to efficiently solve a new layout in under one second. For example, smaller widgets could be contained in larger bounding boxes, and the layout of individual boxes could be solved efficiently in parallel.

Expert Review

We initially considered a comparison to see how ORC layouts perform relative to other methods. Yet, to our knowledge, there are currently no GUI layout systems that can create a single easily "rotatable" layout that works for non-trivial examples. Thus, we started by exploring if designers could work with ORC layouts by performing a qualitative evaluation with nine participants (four female, aged from 21 to 48). All participants were professional UI designers or software engineers from technical companies with substantial experience with GUI layouts and responsive web design.

Procedure. We first showed participants PowerPoint slides with all the figures in this paper and asked them to follow the sequence of edit operations for those figures to familiarize them with our ORC Editor. All the participants found this easy to follow. One stated that " *I think anyone who has ever used a responsive web app will find it fairly intuitive.*" It took only about five minutes for them to finish trying all the edit operations. After having experienced the ORC Editor, most participants were curious about the difference between "responsive design" and ORC layouts. When told that our layout method enables the designer to create only a single layout specification for multiple screen sizes and orientations, a UI designer stated that *"Seems like a great idea. [ORC layouts] could save us a lot of time."*

Then we asked them to create several layouts containing 8-12 widgets in a single container that work well in landscape and portrait mode with the ORC Editor, and interviewed them about their opinions, including how well layouts worked for different screen orientations. In general, they quickly grasped how ORC layout patterns work and were able to create new, rotatable layouts in about 10-20 minutes.

Interviews. In post-hoc interviews, participants expressed generally very positive feelings towards ORC layouts. They found ORC layout editing user-friendly, and many compared their experience with ORC favorably with their day-to-day experience of building responsive UIs. They identified that

Algorithm 1: Flowing widgets around a fixed area.

```
first widget → top left corner
if overlap then
    first widget → after fixed area
end
if current widget is above or below fixed area then
    current_left = prev_right OR current → next_row (hard)
end
if current widget shares any row with the fixed area then
    // break over rows to prevent overlap
    if there is enough space for current widget in this row then
        current_top = prev_top (hard)
        if enough space between previous widget and fixed widget then
            current_left = prev_right (hard)
        end
        else
            current_left = fixed_right (hard)
        end
    end
    else
        current → next_row
        if enough space between previous widget and fixed area then
            current_left = left_boundary (hard)
        end
        else
            current_left = fixed_right (hard)
        end
    end
end
current_top = prev_top (soft with high weight)
current → next_row (soft with low weight)
```

this new layout method complements existing ones and indicated that *"[ORC layouts] allows me to design layouts that are otherwise hard to do. The results of the layout did meet my expectation[s]"*. One stated that *"Usually we need to carefully design different layouts for phones, tablets, and PCs. But [ORC] has great potential to automate [such] repetitive work. [ORC layouts] makes it so much simpler to deal with [mobile device] layouts, as I then need to maintain only one layout"*. Some usability issues of the ORC Editor prototype were criticized, but participants also recognized that integration into an industry-strength GUI editor would address such issues.

# of widgets	# Constraints	Solving Time (s)			
		Insert	Delete	Move	Resize
5	40	0.03	0.02	0.27	0.33
10	85	0.06	0.07	0.36	0.40
15	130	0.24	0.17	0.48	0.49
20	175	0.28	0.29	0.58	0.61
25	220	0.78	0.71	1.39	1.34
30	265	1.69	1.24	1.69	1.82

Table 1: Average solving times with ORC layout (in seconds) for different number of constraints and different operations. ("Resize" in this table means resizing specific widgets.) In our experiments, we used horizontal flow layout with the OR-constraint "to the right OR at the beginning of next line" and the sizes of widgets were determined by their min/pref/max constraints.

7 DISCUSSION

As demonstrated above, OR-constraints enable new ways to specify GUI layouts that can automatically adapt to different screen orientations and sizes. Relative to previously presented layout methods, OR-constraints can express more extensive re-arrangements of widgets. In particular, OR-constraints are able to combine the power of both flow and grid-based layout specifications. A big benefit of ORC layouts is that this method enables the designer to create only a single layout specification that works for multiple screen sizes and orientations. This means that designers do not need to create and maintain multiple layout specifications for each GUI screen.

We explain the trade-off between power and simplicity in the ORC editor as follows: When a user specifies a layout for a GUI container, the system creates the corresponding constraints with default parameters. Through our pre-defined constraint layout patterns, we substantially reduce the complexity presented to the user. Users then only need to modify the parameters and select different options for a layout functionality (but not the constraints themselves) to meet more specialized requirements as desired. By automatically keeping layouts solvable [39], we keep layout editor usability high, as this effectively prevents users from over-specifying a layout.

It is advantageous to create a single layout specification for multiple different (*e.g.*, portrait and landscape) form factors. Similar to challenges in software engineering, it is easy for multiple alternative GUI layout specs to get out of "sync", *e.g.*, when widgets get added/changed/removed. While there may be other ways to deal with this situation, it typically requires decomposing a GUI into many individual building blocks. A single spec is easier to maintain and to keep consistent.

Limitations

One issue with ORC layouts is that for more complex situations a hierarchy may be necessary to solve the constraint system. While a single constraint system offers potentially more flexibility, some naïve forms of prioritization of high-level vs. low-level widgets can make a constraint system time-consuming to solve. As GUI designers are used to decompose complex layouts into a hierarchy we currently do not see this as a major issue. We acknowledge that usability, layout, and aesthetic design principles are not directly supported by the work reported here. If such aspects are expressible as constraints, they can be specified as part of an ORC layout. Yet, we consider this outside the scope of our current work.

8 CONCLUSION

In this paper, we presented ORC layouts, a novel GUI layout method that adds OR-constraints to standard constraint-based layout specifications. ORC layout unifies grid layout and flow layout, offering new possibilities for flexible GUIs that are not supported by any other layout method. Through the use of an efficient SMT solver, ORC layout constraint systems with soft, hard and OR-constraints can typically be solved at interactive rates. We envision that our new layout method could be widely applied in modern web design, document format, and app layout. We plan to open-source our code for future research on OR-constraint layouts.

Z3 is a very general constraint solver. Likely, this generality introduces some computation overhead. As our work is only the first exploration of ORC layouts, we envision that future layout solvers will improve performance further. Thus, we see potential for new, more specialized solving algorithms that can deal with OR-constraints in a more efficient manner.

Acknowledgements

This work builds on initial explorations of OR-constrained layouts by Navid Mohaghegh and the last author.

REFERENCES

[1] Gideon Avrahami, Kenneth P Brooks, and Marc H Brown. 1989. A Two-View Approach to Constructing User Interfaces. *ACM SIGGRAPH Computer Graphics* 23, 3 (1989), 137–146. https://doi.org/10.1007/354054742_40

[2] Greg J. Badros, Alan Borning, and Peter J. Stuckey. 2001. The Cassowary Linear Arithmetic Constraint Solving Algorithm. *ACM Trans. Comput.-Hum. Interact* 8, 4 (2001), 267–306. https://doi.org/10.1145/504704.504705

[3] Thomas Bill, Bertil Lundell, John Alan McDonald, and Michael Sannella. 1992. *Bricklayer: Window Layout Using Linear Programming.* Technical Report. University of Washington, New York, NY, USA.

[4] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. 1992. Constraint Hierarchies. *LISP and Symbolic Computation* 5, 3 (1992), 223–270. https://doi.org/10.1007/978-3-642-85983-_4

[5] Alan Borning, Kim Marriott, Peter Stuckey, and Yi Xiao. 1997. Solving Linear Arithmetic Constraints for User Interface Applications. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology*. ACM, Banff, Alberta, Canada, 87–96. https://doi.org/10.1145/263407.263518

[6] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-_24

[7] James Fogarty and Scott E. Hudson. 2003. GADGET: a Toolkit for Optimization-Based Approaches to Interface and Display Generation. In *Proceedings of the 16th Annual ACM Symposium on User Interface Software and Technology (UIST '03)*. ACM, New York, NY, USA, 125–134. https://doi.org/10.1145/1186562.1015789

[8] Krzysztof Z. Gajos, Daniel S. Weld, and Jacob O. Wobbrock. 2010. Automatically Generating Personalized User Interfaces With Supple, In Proceedings of the 9th International Conference on Intelligent User Interfaces. *Artif. Intell* 174, 12-13, 910–950. https://doi.org/10.1016/j.artint.2010.05.005

[9] Krzysztof Z. Gajos, Daniel S. Weld, and Jacob O. Wobbrock. 2010. Automatically Generating Personalized User Interfaces With Supple. *Artif. Intell* 174, 12-13 (2010), 910–950. https://doi.org/10.1145/964442.964461

[10] Krzysztof Z. Gajos, Jacob O. Wobbrock, and Daniel S. Weld. 2008. Improving the Performance of Motor-Impaired Users With Automatically-Generated, Ability-Based Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*. ACM, New York, NY, USA, 1257–1266. https://doi.org/10.1145/1357054.1357250

[11] Michael Gleicher. 1993. A Graphics Toolkit Based on Differential Constraints. In *Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology*. ACM, New York, NY, USA, 109–120. https://doi.org/10.1145/168642.168653

[12] 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 (UIST '92)*. ACM, New York, NY, USA, 117–124. https://doi.org/10.1145/142621.142635

[13] Hiroshi Hosobe. 2000. A Scalable Linear Constraint Solver for User Interface Construction. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP '02)*. Springer-Verlag, Berlin, Heidelberg, 218–232. https://doi.org/10.1007/3-540-45349-_17

[14] Thibaud Hottelier, Ras Bodik, and Kimiko Ryokai. 2014. Programming by Manipulation for Layout. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, New York, NY, USA, 231–241. https://doi.org/10.1145/2642918.2647378

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

[16] Charles Jacobs, Wilmot Li, Evan Schrier, David Bargeron, and David Salesin. 2003. Adaptive Grid-Based Document Layout. In *ACM SIGGRAPH 2003 Papers (SIGGRAPH '03)*. ACM, New York, NY, USA, 838–847. https://doi.org/10.1145/1201775.882353

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

[18] Christof Lutteroth, Robert Strandh, and Gerald Weber. 2008. Domain Specific High-Level Constraints for User Interface Layout. *Constraints* 13, 3 (2008), 307–342. https://doi.org/10.1145/1496976.1496977

[19] 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 (AUIC '06). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 53–60. https://doi.org/10.1007/978-1-4842-2662-_20

[20] Christof Lutteroth and Gerald Weber. 2008. Modular Specification of GUI Layout Using Constraints. In *2008. ASWEC 2008. 19th Australian Conference on Software Engineering*. IEEE, New York, NY, USA, 300–309. https://doi.org/10.1109/ASWEC.2008.4483218

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

[22] Brad Myers, Scott E. Hudson, Randy Pausch, and Randy Pausch. 2000. Past, Present, and Future of User Interface Software Tools. *ACM Trans. Comput.-Hum. Interact* 7, 1 (2000), 3–28. https://doi.org/10.1177/154193120004400206

[23] Brad A. Myers. 1995. User Interface Software Tools. *ACM Transactions on Computer-Human Interaction* (1995). https://doi.org/10.1145/200968.200971

[24] Brad A. Myers and William Buxton. 1986. Creating Highly-Interactive and Graphical User Interfaces by Demonstration. *SIGGRAPH Comput. Graph* 20, 4 (1986), 249–258. https://doi.org/10.1145/15922.15914

[25] Brad A Myers, Richard G McDaniel, Robert C Miller, Alan S Ferrency, Andrew Faulring, Bruce D Kyle, 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, 6 (1997), 347–365.

[26] Robert Nieuwenhuis and Albert Oliveras. 2006. On SAT Modulo Theories and Optimization Problems. In *Theory and Applications of Satisfiability Testing - SAT 2006*. Springer, Berlin, Heidelberg, 156–169. https://doi.org/10.1007/1181494_18

[27] Pavel Panchekha and Emina Torlak. 2016. Automated Reasoning for Web Page Layout. *SIGPLAN Not* 51, 10 (2016), 181–194. https://doi.org/10.1145/2983990.2984010

[28] Erica Sadun. 2013. *iOS Auto Layout Demystified*. Addison-Wesley Professional, Boston, US.

[29] 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 (EICS '13)*. ACM, Gothenburg, Sweden, 275–284. https://doi.org/10.1145/3197231.3197249

[30] 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, Toronto, ON, Canada, 642–647. https://doi.org/10.1016/j.infsof.2015.10.005

[31] Gurminder Singh, Chun Hong Kok, and Teng Ye Ngan. 1990. Druid: a System for Demonstrational Rapid User Interface Development. In *Proceedings of the 3rd Annual ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST '90)*. ACM, New York, NY, USA, 167–177. https://doi.org/10.1145/97924.97943

[32] Nishant Sinha and Rezwana Karim. 2015. Responsive Designs in a Snap. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, Bergamo, Italy, 544–554. https://doi.org/10.1145/2786805.2786808

[33] John M Vlissides and Steven Tang. 1991. A Unidraw-Based User Interface Builder. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology*. ACM, Hilton Head, South Carolina, 201–210. https://doi.org/10.1145/120782.120804

[34] 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, Washington, DC, USA, 67–74. https://doi.org/10.1109/ASWEC.2010.38

[35] Daniel S. Weld, Corin Anderson, Pedro Domingos, Oren Etzioni, Krzysztof Gajos, Tessa Lau, and Steve Wolfman. 2003. Automatically Personalizing User Interfaces. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1613–1619. http://dl.acm.org/citation.cfm?id=1630659.1630944

[36] 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 (CHI '90)*. ACM, Seattle, Washington, USA, 27–34. https://doi.org/10.1007/978-3-319-67744-_2

[37] Brad Vander Zanden and Brad A Myers. 1991. The Lapidary Graphical Interface Design Tool. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 465–466. https://doi.org/10.1145/108844.109005

[38] Clemens Zeidler, Christof Lutteroth, Wolfgang Stuerzlinger, and Gerald Weber. 2013. Evaluating Direct Manipulation Operations for Constraint-Based Layout. In *IFIP Conference on Human-Computer Interaction (INTERACT)*. Springer, Berlin, Heidelberg, 513–529. https://doi.org/10.1007/978-3-642-40480-1_35

[39] Clemens Zeidler, Christof Lutteroth, Wolfgang Sturzlinger, and Gerald Weber. 2013. The Auckland Layout Editor: An Improved GUI Layout Specification Process. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology (UIST '13)*. ACM, St. Andrews, Scotland, United Kingdom, 343–352. https://doi.org/10.1145/2379256.2379287

[40] Clemens Zeidler, Christof Lutteroth, and Gerald Weber. 2012. Constraint Solving for Beautiful User Interfaces: How Solving Strategies Support Layout Aesthetics. In *Proceedings of the 13th International Conference of the NZ Chapter of the ACM's Special Interest Group on Human-Computer Interaction*. ACM, New York, NY, USA, 72–79. https://doi.org/10.1145/2379256.2379268

[41] Clemens Zeidler, Johannes Müller, Christof Lutteroth, and Gerald Weber. 2012. Comparing the Usability of Grid-Bag and Constraint-Based Layouts. In *Proceedings of the 24th Australian Computer-Human Interaction Conference (OzCHI '12)*. ACM, New York, NY, USA, 674–682. https://doi.org/10.1145/2414536.2414638

[42] Clemens Zeidler, Gerald Weber, Alex Gavryushkin, and Christof Lutteroth. 2017. Tiling Algebra for Constraint-Based Layout Editing. *Journal of Logical and Algebraic Methods in Programming* 89 (2017), 67–94. https://doi.org/10.1016/j.jlamp.2017.01.004