ORCSolver: An Efficient Solver for Adaptive GUI Layout with OR-Constraints

Yue Jiang[1] **Wolfgang Stuerzlinger**[2] **Matthias Zwicker**[1] **Christof Lutteroth**[3]
[1]Department of Computer Science, University of Maryland, College Park, MD, USA
[2]School of Interactive Arts + Technology (SIAT), Simon Fraser University, Vancouver, BC, Canada
[3]Department of Computer Science, University of Bath, Bath, UK
{yuejiang, zwicker}@cs.umd.edu w.s@sfu.ca c.lutteroth@bath.ac.uk

Figure 1. ORCSolver is able to adapt a layout between different sizes and orientations at near-interactive rates, based on a single layout specification. The layout is adjusted to fit the aspect ratio, and the optional "CHI2019" logo and buttons 1, 2, 3 are automatically removed as space gets scarce.

ABSTRACT
OR-constrained (ORC) graphical user interface layouts unify conventional constraint-based layouts with flow layouts, which enables the definition of flexible layouts that adapt to screens with different sizes, orientations, or aspect ratios with only a single layout specification. Unfortunately, solving ORC layouts with current solvers is time-consuming and the needed time increases exponentially with the number of widgets and constraints. To address this challenge, we propose ORCSolver, a novel solving technique for adaptive ORC layouts, based on a branch-and-bound approach with heuristic preprocessing. We demonstrate that ORCSolver simplifies ORC specifications at runtime and our approach can solve ORC layout specifications efficiently at near-interactive rates.

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

CCS Concepts
•**Human-centered computing** → **User interface toolkits;**

INTRODUCTION
Constraint-based layout models have been widely used in many graphical user interfaces (GUIs) for applications because they are more flexible and powerful than other layout models. For instance, Apple's AutoLayout [59] adapts interfaces on different devices ranging from desktops to smartphones and CSS's Flex(ible) Box[1] uses constraints to fit the content and solve alignment problems dynamically. Constraint-based layout models can align widgets across different groups, which is impossible with grid layouts. Yet, common constraint-based layout models are not without limitations. They cannot support flow layouts, and despite their flexibility, device diversity has been a long-term challenge for them. Even with constraint-based layouts, separate layout specifications need to be defined for different sizes, orientations (portrait and landscape), or aspect ratios of the screens. Such multiple specifications can be hard to maintain and to keep consistent.

To address these problems, Jiang et al. [38] proposed an approach for constraint-based graphical user interface layouts based on OR-constraints (ORC). An OR-constraint is a disjunction of multiple constraints, where only one needs to be true. ORC layouts specify adaptive GUIs for different devices, as they can adapt a GUI to screens with different sizes, orientations, and aspect ratios with predictable results, using only a single layout specification. This has the potential to make GUI design more efficient and less error-prone, as designers do not need to manually synchronise changes between different layout specifications for different devices. However, solving ORC layouts is currently slow, due to the computational complexity involved. Solving OR- and soft constraints is more challenging than solving hard constraints due to the combinatorial explosion. Modern constraint solvers such as Z3 [10] cannot handle OR-constraints efficiently, as they need to eval-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CHI '20, April 25–30, 2020, Honolulu, HI, USA.
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6708-0/20/04 ...$15.00.
http://dx.doi.org/10.1145/3313831.3376610

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

uate very large numbers of potential combinations. Although Jiang et al. [38] stated that ORC specifications can be solved interactively, the solving time in their experiments increased dramatically with the number of widgets. For example, their approach took almost 2 seconds to solve a layout with only 30 widgets.

Here, we propose ORCSolver – a novel efficient solving technique for adaptive GUI layout with OR-constraints. ORCSolver is the first solver that can solve complex, realistic layouts made of linear constraints, flows, and their combinations at near-interactive rates. ORCSolver uses a heuristic optimization algorithm to reduce the computational complexity of solving adaptive GUI layouts with OR-constraints. Instead of feeding many OR- and soft constraints directly into a general-purpose solver, such as Z3 [10], we use a branch-and-bound algorithm with efficient heuristics to reduce the solution space and discard branches based on bounding criteria. We then produce a smaller set of constraints, which represent the same layout as the original set but are tailored to the current screen size and have thus drastically reduced complexity. We demonstrate that our solving technique can significantly reduce the solving time for adaptive GUI layouts with OR-constraints, making the following contributions:

- A formal notation for ORC layouts, which can be fed into our ORCSolver and help to formalise solving strategies.
- A novel solving technique for ORC layouts, based on a branch-and-bound approach with interval arithmetic and modular heuristics, which drastically reduces solving time.
- A set of heuristics to reduce the number of constraints in ORC layout specifications during solving. We formalize the solver as a framework with pluggable heuristics.
- Experimental evidence of the efficiency of our approach through comparison with three other solvers, which demonstrates that ORC layouts can be solved much more efficiently (*e.g.*, more than two orders of magnitude faster than Z3 with more than 100 widgets).

BACKGROUND

Linear constraint layouts have been widely used for responsive web design [28, 43] and mobile apps [60, 71]. Constraint-based layouts are easier to maintain compared to programming approaches [49, 50] and overcome common limitations of other layout models, such as an inability to align widgets across containment hierarchies [42] and maintenance issues [41, 76]. Constraint-based layout specifications are usually systems of linear equations and inequalities, being either hard or soft constraints.

Soft and Hard Linear Constraints: Hard constraints are ones which must be satisfied, while soft constraints can be neglected if not all constraints can be satisfied simultaneously. Soft constraints form a hierarchy, where each has a weight to define its priority [6]. Higher weights are given to more important widgets, which implies higher priority. Hard constraints are soft constraints with infinite weights.

OR-Constraints: OR-constraints are disjunctive, where the entire constraint functions as a hard one, while each disjunctive part is a soft constraint. Only one disjunctive part needs to be satisfied to make the OR-constraint true. OR-constraints are more challenging to solve than hard constraints. For OR-constraints, solvers typically need to check multiple cases to identify a branch in each disjunctive constraint that can be satisfied, while for soft constraints, their attached weights necessitate decisions about which ones should be neglected and which ones satisfied.

Interval Arithmetic: Interval arithmetic is useful for solving systems of equations and inequalities [54]. This numerical method puts bounds on each variable to identify ranges that contain promising solutions, to quickly identify valid solutions for complex optimization problems. When we apply a function f to an uncertain value x, the result is uncertain since the input x is indeterminate. Thus, we do not know what the result is or how far we are from it. Instead of using x as the input, we work with an interval $[a,b]$ that contains x. Since all its operations are closed, after we apply the function f to the interval $[a,b]$ with interval arithmetic, the resulting interval $[c,d]$ must contain the accurate value of $f(x)$. We use interval arithmetic to rule out impossible or ineffective solutions. Once we know that the best solution must be in an interval $[c,d]$, we can rule out all the potential solutions outside of that interval.

Branch & Bound (B&B): The B&B algorithm [44] is a well-known approach to solving combinatorial optimization problems. This search algorithm explores the branches of a rooted tree. Each branch contains a subset of the solution set. With this, B&B recursively divides the search space into smaller subspaces and minimizes an objective function on all of them. Once it finds that a branch cannot contain an optimal solution, it discards that branch and does not explore its subtree further. Zeidler et al. [78] used B&B to automatically generate layouts for alternative screen orientations. They investigated only mobile device rotation, but not different screen sizes. Other layout generation approaches such as SUPPLE [23] also used B&B. All these approaches suffer from combinatorial explosion.

Quadratic Programming (QP) Solvers: QP problems [15] have been widely explored. Most QP solvers are based on three approaches: active set methods [70] (*e.g.,* qpOASES [13]), interior point methods [53] (*e.g.,* MOSEK [1], Gurobi [35], CVXGEN [48], OOQP [26]), and first order methods [15] (*e.g.,* SCS [56]). The alternating approach of multipliers (ADMM) [8], a first order method, is widely used in practice for solving QP with good convergence behavior. The OSQP solver [4] is a state of the art QP solver using an improved ADMM method [8] to avoid strong dependencies on the problem data.

Solving Approaches for OR-Constraints: Previous layout solvers [3, 7, 16, 30, 31, 37, 45, 47, 61, 62] can only solve either linear constraint-based layouts or flows but not both, *i.e.,* cannot solve systems with OR-constraints. Disjunctive constraints for GUIs were first proposed for non-overlap, *i.e.*, to ensure widgets never overlap [46]. However, the proposed method cannot solve disjoint disjunctions and works only if solutions can transition continuously between disjuncts without passing through unsatisfiable regions. The ORC layout approach [38] was the first to use the Z3 solver to deal with general GUI layouts with various types of OR-constraints. The

Z3 solver [10] is a satisfiability modulo theory (SMT) solver that is able to solve specifications with disjunctive constraints. It has been previously used for solving CSS layout specifications [57] and layout editing [33]. Z3 can handle both disjoint and non-disjoint disjunctions, which enables this approach to unify constraint-based and flow layouts. However, both these approaches suffer from runtime issues.

RELATED WORK

GUI Builders

GUI builders support designers to specify and generate layouts. FormsVBT [2] introduced a textual representation and an interactive editor for interfaces. Bramble [27] applied differential constraints to generate graphical editing applications. Gilt [29] used Graphical Tabs and styles to simplify layout specifications. Amulet [52] allowed developers to combine some properties of flow and grid layouts programmatically. OPUS [34] supported direct manipulation interfaces with a graphical notation. Ibuild [67] allowed users to modify simulations of layouts. Peridot [51], Druid [64], and Lapidary [72] created code automatically based on users' demonstration. UI Façades [66] afforded direct manipulation to manually reconfigure and recombine existing interfaces. Modern GUI builder approaches can create layout constraints robustly based on direct manipulation [63, 74, 68]. Based on intersections of objects, Rockit [39] determined graphical constraints in a 2D scene. Most layouts can be specified as constraint systems [77], and designing a good constraint system is important for adaptive GUI layouts: an ambiguous specification with too few constraints ('underspecification') can lead to unpredictable results, while too many ('overspecification') can lead to conflicts, lack of valid solutions, and much increased solving times.

UI Generators

Some work proposed to improve GUI customization and adaptation [69]. Fogarty et al. [14] supported optimization for interface generation. SUPPLE [22, 23] automatically generated user interfaces by minimizing interface operations to meet screen-size constraints. It was also used to generate GUIs for Ubicomp apps [17, 20], improve personalization, maintain consistency [21], and generate interfaces for people with physical disabilities [11, 18, 24, 25]. Arnauld [19] learned and generated the parameters of cost functions in optimization-based systems. Other tools generated layout alternatives through templates [36] or modifiable suggestions [65, 71].

Layout Solvers

Given a constraint system, a layout solver is needed to determine a solution for widget positions and sizes. There are various solvers for constraint-based GUI specifications, using linear or quadratic programming [3, 5, 7, 30, 40, 73]. They use objective functions to minimize deviations from an optimal solution, which can improve aesthetics [75].

CLP(R) [37] includes constraints in logic programming for linear equalities and inequalities. It gradually optimizes the variables, but is not suitable for interactive use. DeltaBlue [16, 62] and Skyblue [61] are local propagation constraint solvers and can handle constraint hierarchies, but cannot solve

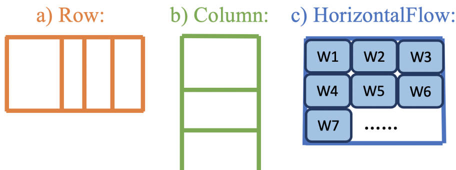


Figure 2. a) A Row is a rectangular horizontal arrangement of sub-layout, and b) a Column a rectangular vertical one. c) Horizontal flow.

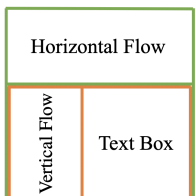


Figure 3. ORC layout defined as: *Column(HorizontalFlow, Row(VerticalFlow, TextBox))*

simultaneous constraints. Cassowary [3, 7], an incremental constraint solver for user interface applications, solves linear equalities and inequalities with an incremental simplex approach. QOCA [7, 45, 47] applies tableau-based methods to solve constraint hierarchies. HiRise [30] and HiRise2 [31] use a simplex method with an LU decomposition-based algorithm and employ ordered constraint hierarchies to solve linear equality and inequality constraints.

These solvers use different penalty functions (comparators) to handle soft constraints. For example, Cassowary [7, 3] uses weighted-sum-better, QOCA [7, 45, 47] uses least-squares-better, and HiRise [30] and HiRise2 [31] use locally-error-better to solve constraint hierarchies. These solvers support linear constraints with priorities and aim to satisfy as many constraints as possible, subject to priorities.

FORMAL NOTATION

Constraints are low level and hard to specify manually [74]; in most cases, layouts can be expressed in a more abstract and developer-friendly manner [77]. We provide a formal notation and corresponding application programming interface (API)[2] to enable GUI developers to specify ORC layouts that can be solved at near-interactive rates by our solver. Similar to other GUI toolkits, our notation's layout specifications consist of recursively nested, rectangular layouts that are all subtypes of a generic layout supertype. Layouts can be simple widgets, or full-fledged, modular constraint-based layout specifications [42]. Similar to other layout models [32], we formalise fixed horizontal and vertical alignments as *Row* and *Column* layout types (Figure 2 a, b), and flow layouts as *HorizontalFlow* (*HF* for short) (Figure 2 c) and *VerticalFlow* (*VF* for short). All ORC layout patterns are similarly represented as layout types. For example, the layout in Figure 3 can be formalized as *Column(HorizontalFlow, Row(VerticalFlow, TextBox))*. The layout uses a *Column* layout dividing the window into a horizontal flow and the lower part, which is a *Row* layout containing a vertical flow and a text box.

[2]The full API is included in the open source repository.

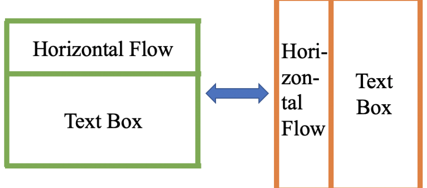


Figure 4. ORC layout defined as: *Pivot(Column(HorizontalFlow, TextBox))*

Pivot Layout: We define a *Pivot* layout to encode more flexibility into the layout design. The *Pivot* layout allows the solver to turn horizontal arrangements into vertical ones, and vice versa, using OR constraints. By default, the *Pivot* layout modifies only its first sub-layout. For example, the layout in Figure 4 can be formalized as *Pivot(Column(HorizontalFlow, TextBox))*. The *HorizontalFlow* could, *e.g.*, contain toolbar widgets. By using *Pivot*, the *Column* can be turned into a *Row* to adapt from a landscape orientation ('toolbar at the top') to a portrait orientation ('toolbar on the left'). Internally, the solver breaks the layout down into two alternatives: *Column(HorizontalFlow, TextBox)* OR *Row(HorizontalFlow, TextBox)*. Figure 1 shows a more complex example with the specification *Pivot(Column(HorizontalFlow1, Column(HorizontalFlow2, Pivot(Column(HorizontalFlow3, HorizontalFlow4)))))*, with the "CHI2019" logo and the buttons 1, 2, 3 defined as optional in the layout. The first *Pivot* controls the numbered buttons, and the second *Pivot* independently controls the bottom input fields. If found to be more optimal by the solver, the first *Pivot* could move the numbered buttons to the left while the second *Pivot* does not change the input fields (Figure 1 middle → left), or the first *Pivot* could keep the numbered buttons left while the second *Pivot* turns the input fields (Figure 1 middle → right).

Scope: Our formal notation can represent any rectangular tiling layout, *i.e.*, any grid-like subdivision through linear constraints [68, 77]. Common layouts such as box, grid, and flow are directly represented with our formal notation, and low-level linear constraints can be added where necessary, *e.g.,* for alignment across sub-layout boundaries and row-/col-span. In contrast to other layout notations [3, 7, 45, 47, 74, 77] and beyond simple resizing or flows, our notation can also specify how layouts change in response to changing UI sizes.

SOLVING OVERVIEW

ORCSolver is a conceptually new approach, integrating branch-and-bound, greedy optimization, and quadratic programming, which specifically addresses multi-device UI layouts requiring a particularly high degree of solving flexibility not supported by other solvers, and offers for the first time multi-device layout solving at near-interactive rates. ORCSolver is a general contribution as it can be used for any layout that can be represented with linear constraints and/or flows, which includes almost all commonly used layout models and document layout patterns [38, 68, 74]. Some rare layouts are currently not supported, such as overlapping elements [77].

Instead of feeding a layout specification with a potentially large number of OR- and soft constraints directly into a constraint solver, ORCSolver uses a B&B algorithm to keep track of potential solutions for all the sub-layouts and prune branches based on bounding criteria. It also uses modular heuristics for each layout type to reduce the number of OR constraints by adapting sub-layouts to the given layout context, *e.g.*, the given available size. The result is a much smaller set of constraints, representing an optimal or near-optimal adaptation of the original constraint system to the given layout context. The resulting system can then be solved much more quickly than the original using an off-the-shelf solver.

Branch & Bound (B&B)

To start, ORCSolver is given a layout specification in our formal notation. The specification can contain high-level layout types as well as low-level constraints. ORCSolver then searches the layout space with B&B by considering feasible subsets of constraints for each sub-layout and ruling out whole branches by considering hard constraints and interval bounds of the objective function. To speed up solving, ORCSolver prioritizes branches that are likely to lead to optimal solutions.

Selection of Sub-layout Solving Order

To reduce assumptions and speed up B&B, sub-layouts that are more constrained in their size are processed first. This is done by prioritising sub-layouts according to their number of *firm edges*, *i.e.*, edges that cannot move (much) in the layout. As ORCSolver optimizes a layout, it defines firm edges iteratively. Initially, the layout boundaries are the only firm edges. As ORCSolver processes sub-layouts, their edges become firm as they are placed in the layout with more certainty. Sub-layouts are preferentially resolved or simplified with greedy algorithms if they have a high number of firm edges, which increases certainty and helps narrow down the search space. For example, consider a *Row* at the top of an ORC layout: the leftmost sub-layout has at least two firm edges (top and left). Once ORCSolver processes the leftmost sub-layout, new firm edges are generated around that sub-layout and the next sub-layout to the right has now also at least two firm edges.

Greedy Algorithms

During B&B, the aforementioned heuristics are applied to the sub-layouts of an ORC layout. Our heuristics are algorithms that reduce the number of OR-constraints for the different layout types. Optimizing complex sub-layouts, such as flows, with brute-force B&B search leads to combinatorial explosion. Our heuristics make the B&B approach computationally tractable, which then also assures an overall optimal layout.

ORCSolver uses heuristics for the different ORC layout subtypes that reduce the number of OR- and soft constraints by either removing them or changing them into hard constraints, while at the same time ensuring that the simplified constraint system is equivalent (or near-equivalent) to the original one. For example, horizontal flow layouts use an OR-constraint "to the right of the previous widget OR at the beginning of the next row", which is necessary for generality but introduces one OR-constraint per involved widget. If ORCSolver can estimate which widgets should be placed into which rows during our constraint reduction phase, it can simply assign either a "to the right of the previous widget" or an "at the beginning of the next row" constraint to each widget. Both of these are hard

constraints and can be solved easily. Another example is that each widget typically has minimum, preferred, and maximum sizes. Typically, these are implemented as hard constraints that make sure the widget size is larger than or equal to the minimum size and smaller than or equal to the maximum size, and also a soft constraint to preferably give it its preferred size. In many cases, ORCSolver can estimate the final size of a widget during our heuristic preprocessing, which means it can then replace all the size constraints of a widget with two simple hard constraints that set its appropriate size.

Solver Selection

When all sub-layouts have been processed with heuristics and OR- and soft constraints have been removed as much as possible, ORCSolver has reached a leaf node of the B&B search tree. Before backtracking to process sub-layouts with different parameters, ORCSolver finishes solving the overall layout by applying a standard solver on the reduced specification. If several OR-constraints could not be eliminated, then ORCSolver uses the Z3 solver as it can efficiently solve general disjunctive constraints. If all OR-constraints could be eliminated, ORCSolver uses a simpler linear solver such as Cassowary [3] or a quadratic solver such as OSQP [4]. If very few OR-constraints remain, ORCSolver traverses all possible alternatives using B&B and solves them with the simpler solver, as for few alternatives this is often faster than using a more general solver such as Z3.

OPTIMIZING FLOW LAYOUTS

ORC layout unifies constraint-based layout and flow layout [38]. Optimisation of flow layouts is a central part of ORCSolver that addresses many ORC layout patterns. In the following we describe the heuristic algorithm ORCSolver uses to locally optimise sub-layouts involving flow in linear time, in order to reduce their constraints. Without loss of generality and to keep the description simple, we discuss only horizontal flows as vertical flows are analogous. Furthermore, although we call the elements arranged in flows 'widgets' as this is the most common use case, they could be other sub-layouts, which means the approach might be applied recursively.

Given an estimate of width based on firm edges, ORCSolver first uses heuristics to compute all possible numbers of rows in a horizontal flow, and then, given a specific number of rows, ORCSolver computes the optimal arrangement of widgets in the flow. ORCSolver can estimate minimum, preferred, and maximum numbers of rows for a flow layout quickly by considering all contained widgets in their minimum, preferred, and maximum widths, respectively. This range of possible numbers of rows is then explored in the B&B process by repeatedly applying algorithm 1 with different *numRows* values. Using a simple greedy heuristic, ORCSolver tiles one widget after another until it hits a firm edge, and then move on to the next row. To fill a row without leaving gaps, ORCSolver redistributes the sizes of widgets evenly over the available width to create a balanced appearance [75].

Objective Function

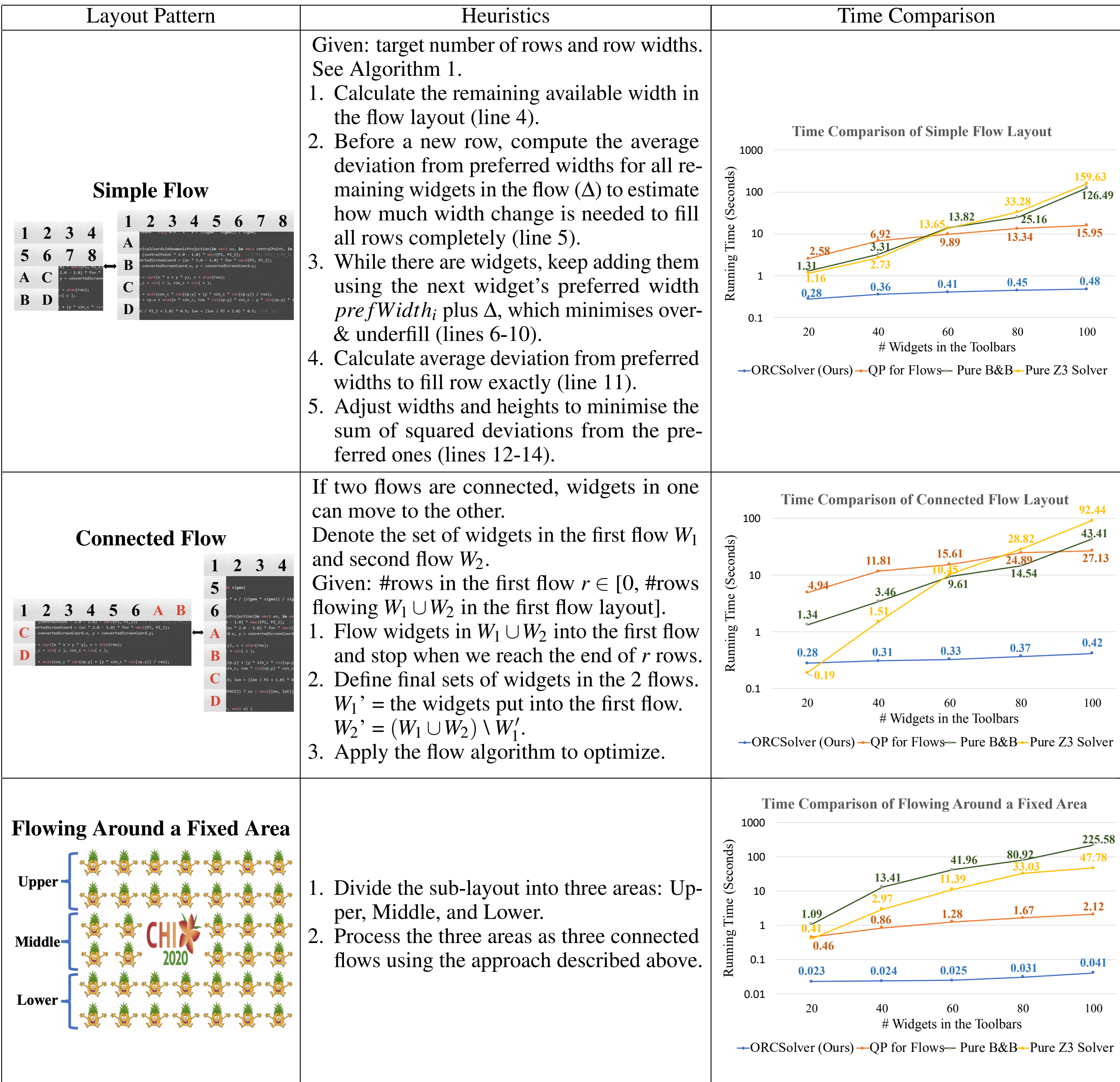
To create a 'balanced' layout appearance [75], our objective function for flows is the sum of the L^2 loss of the deviations between preferred and resulting sizes, resulting in a least-squares approach that reduces deviations by minimising their squares. Further, deviations from the preferred widget sizes can be scaled by widget weights:

Algorithm 1: Greedy Flow Optimisation

1. $i \leftarrow 0$ `// widget index`
2. **for** $r \leftarrow 1$ **to** *numRows* **do**
3. $\quad start \leftarrow i$ `// index of first widget in row`
   $\quad$ `// remaining available layout width`
4. $\quad width_{totalAvail} \leftarrow (numRows - r) \times width_{row}$
   $\quad$ `// avg. deviation from pref. width`
5. $\quad \Delta \leftarrow \dfrac{width_{totalAvail} - \sum_{j \geq start}(width_j)}{numWidgets - start}$
   $\quad$ `// keep adding widgets while we get closer to row width`
6. $\quad width_{rowAvail} \leftarrow width_{row}$
7. $\quad$ **while** $i < num_widgets$ *AND* $|width_{rowAvail} - (prefWidth_i + \Delta)| < |width_{rowAvail}|$ **do**
8. $\quad\quad width_{rowAvail} \leftarrow width_{rowAvail} - (prefWidth_i + \Delta)$
   $\quad\quad i \leftarrow i + 1$
9. $\quad$ **end**
10. $\quad end \leftarrow i - 1$ `// index of last widget in row`
    $\quad$ `// avg. deviation from pref. width for row`
11. $\quad \Delta_{row} \leftarrow (width_{row} - \dfrac{\sum_{j=start..end} prefWidth_j}{end - start + 1}$
    $\quad$ `// redistribute width in row to fit and set height minimising squared deviations`
12. $\quad$ **for** $j \leftarrow start$ **to** end **do**
13. $\quad\quad width_j \leftarrow prefWidth_j + \Delta_{row}$
14. $\quad\quad height_j \leftarrow \dfrac{\sum_{k=start..end}(prefHeight_k)}{end - start + 1}$
15. $\quad$ **end**
16. **end**

$$loss = \sum_i weight_i \times ((width_i - prefWidth_i)^2 + (height_i - prefHeight_i)^2) \quad (1)$$

Instead of considering min/max constraints as hard constraints, we consider them as soft ones with very large weights. We add terms for them to the loss function, in a manner similar to the squared deviations from preferred sizes. This allows widgets to go beyond min/max sizes to avoid gaps in rows or prevent infeasible layouts, but incurs high penalties to make sure this is only done if there is no other solution. For the optional widget pattern, described below, we assigned a loss based on the omitted widgets' weights. This ensures that the layout contains as many of those widgets as possible.



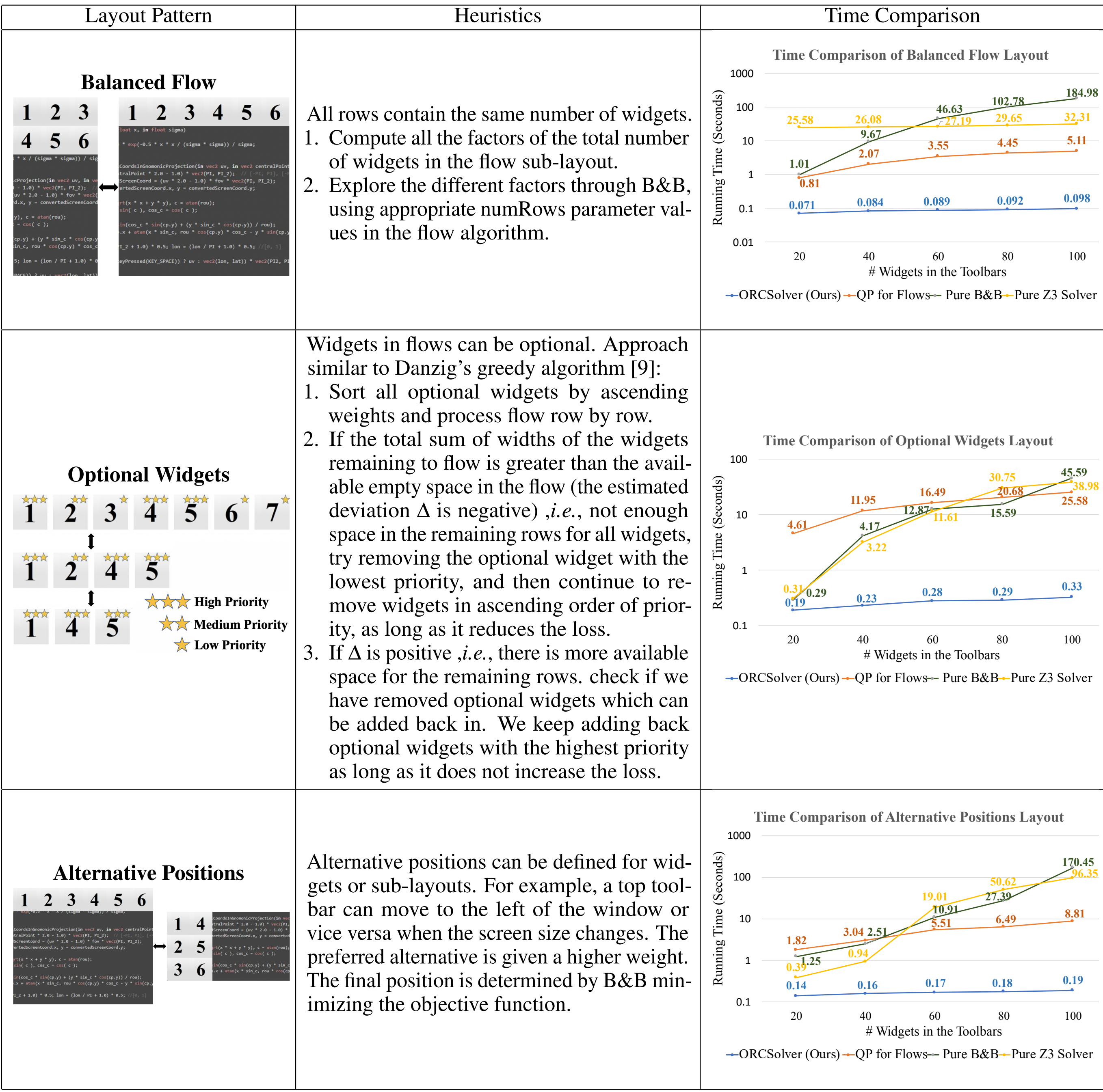
Layout Pattern	Heuristics	Time Comparison
Simple Flow	Given: target number of rows and row widths. See Algorithm 1. 1. Calculate the remaining available width in the flow layout (line 4). 2. Before a new row, compute the average deviation from preferred widths for all remaining widgets in the flow (Δ) to estimate how much width change is needed to fill all rows completely (line 5). 3. While there are widgets, keep adding them using the next widget's preferred width $prefWidth_i$ plus Δ, which minimises over- & underfill (lines 6-10). 4. Calculate average deviation from preferred widths to fill row exactly (line 11). 5. Adjust widths and heights to minimise the sum of squared deviations from the preferred ones (lines 12-14).	Time Comparison of Simple Flow Layout
Connected Flow	If two flows are connected, widgets in one can move to the other. Denote the set of widgets in the first flow W_1 and second flow W_2. Given: #rows in the first flow $r \in [0,$ #rows flowing $W_1 \cup W_2$ in the first flow layout]. 1. Flow widgets in $W_1 \cup W_2$ into the first flow and stop when we reach the end of r rows. 2. Define final sets of widgets in the 2 flows. W_1' = the widgets put into the first flow. W_2' = $(W_1 \cup W_2) \setminus W_1'$. 3. Apply the flow algorithm to optimize.	Time Comparison of Connected Flow Layout
Flowing Around a Fixed Area	1. Divide the sub-layout into three areas: Upper, Middle, and Lower. 2. Process the three areas as three connected flows using the approach described above.	Time Comparison of Flowing Around a Fixed Area

Table 1. Comparison between ORCSolver and other approaches for simple, connected flow, and flowing around a fixed area.

GLOBAL OPTIMISATION USING BRANCH & BOUND

ORCSolver constructs the B&B tree for a layout based on its formal ORC layout notation. The tree is initially constructed as a one-branch tree, containing sub-layouts in the order in which they are chosen for processing based on their firm edges. For instance, the tree for the layout in Figure 3 with the notation *Column(HorizontalFlow, Row(VerticalFlow, TextBox))* can be represented as shown in Figure 6 a). Similarly, the tree for the layout in Figure 4 with the formalization *Pivot(Column(HorizontalFlow, TextBox))* can be represented as shown in Figure 5 a).

Starting from the root node, as ORCSolver goes down the tree, it adds more and more constraints to the linear system. Each node in the B&B tree has a layout specification containing variables, constraints and objectives. As ORCSolver goes down the tree, it clones the specification from the parent node and add simplified variables, constraints, and objectives for the current processed sub-layout based on the preprocessing results (*e.g.,* the heuristics for flows). ORCSolver solves the specification using a quadratic solver in the leaf node to calculate its overall loss value. If a node still contains OR-constraints after simplification, the Z3 solver is used for the solving step. Whenever ORCSolver reaches a node with a



Layout Pattern	Heuristics	Time Comparison
Balanced Flow	All rows contain the same number of widgets. 1. Compute all the factors of the total number of widgets in the flow sub-layout. 2. Explore the different factors through B&B, using appropriate numRows parameter values in the flow algorithm.	Time Comparison of Balanced Flow Layout
Optional Widgets High Priority Medium Priority Low Priority	Widgets in flows can be optional. Approach similar to Danzig's greedy algorithm [9]: 1. Sort all optional widgets by ascending weights and process flow row by row. 2. If the total sum of widths of the widgets remaining to flow is greater than the available empty space in the flow (the estimated deviation Δ is negative) ,*i.e.*, not enough space in the remaining rows for all widgets, try removing the optional widget with the lowest priority, and then continue to remove widgets in ascending order of priority, as long as it reduces the loss. 3. If Δ is positive ,*i.e.*, there is more available space for the remaining rows. check if we have removed optional widgets which can be added back in. We keep adding back optional widgets with the highest priority as long as it does not increase the loss.	Time Comparison of Optional Widgets Layout
Alternative Positions	Alternative positions can be defined for widgets or sub-layouts. For example, a top toolbar can move to the left of the window or vice versa when the screen size changes. The preferred alternative is given a higher weight. The final position is determined by B&B minimizing the objective function.	Time Comparison of Alternative Positions Layout

Table 2. Comparison between ORCSolver and other approaches for balanced flow, optional widgets layout, and alternative positions patterns.

larger penalty value than the currently best leaf, it removes the entire sub-tree rooted at that node since the loss increases further as it goes down the tree, with more sub-layouts being processed to contribute to that loss. When hitting a leaf node, ORCSolver compares its loss with the currently best solution. If smaller, then the leaf is stored as the new best solution.

Figure 5 shows the B&B tree when ORCSolver solves the layout in Figure 4, *i.e.*, *Pivot(Column(HorizontalFlow, TextBox))*. The root contains the original layout formalization. The *Pivot* layout contains two alternatives, *i.e.*, *toolbar at the top* or *toolbar on the left*. The solver uses B&B to process the first alternative, starting with the *toolbar at the top*, which means the solver adds a child node to the root node containing a simplified specification, *i.e.*, *Column(HorizontalFlow, TextBox)*. The solver then moves on to the next level of the tree, the *HorizontalFlow*, which contains a set of toolbar widgets. The solver applies the heuristic solving module for the *HorizontalFlow* and adds a new child node with a simplified specification, *i.e.*, *Column(heuristics result for HorizontalFlow, TextBox)*. The solver also computes the loss of this node according to the objective function. Then the solver moves on to process the

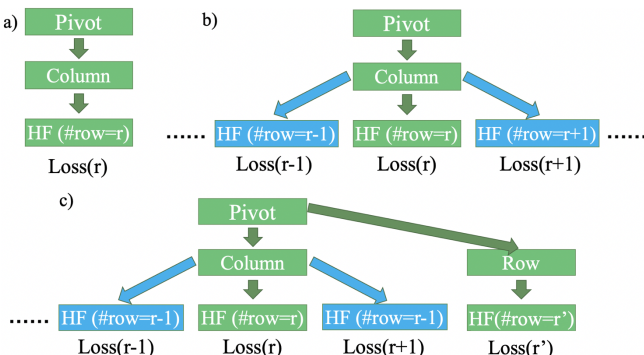


Figure 5. Solving process for Figure 4's layout. The *Pivot* layout has two alternatives, which ORCSolver processes individually to find the best solution.

next level for the *TextBox*, which is a leaf node of the solving tree. The layout does not contain any OR-constraints at this node, so ORCSolver can use a normal quadratic programming solver to calculate the final loss value for the layout. Once it reaches a leaf node, a global variable is used to maintain the currently best solution with the smallest loss value for the whole layout. Whenever the loss of a reached node is larger than the penalty of the currently best solution, ORCSolver ignores the entire subtree of that node. Then it backtracks up to the *Pivot* node and process the other alternative, *i.e.*, *toolbar on the left*. ORCSolver processes this branch as before. Whenever it reaches the bottom of the tree, ORCSolver checks whether it has smaller penalty value than the currently best solution and if so, overwrite the best solution.

Discrete Gradient Search

As part of the B&B process, the solver conducts a gradient search for better solutions based on parameters of the heuristics modules, such as the number of rows or columns in flows. For example, as shown in Figure 6, when ORCSolver solves the layout in Figure 3, the solver first processes the horizontal flow sub-layout since it is the first sub-layout in the *Column* layout. It starts with the preferred number of rows ($numRows_{pref}$) to process the horizontal flow and get a good local fit. Then it moves on to the next sub-layout in the *Column* which is the *Row*. ORCSolver processes the *Row* from left to right, so the next sub-layout to process is the vertical flow. Similarly, it starts with the preferred number of columns ($numCols_{pref}$). Finally, it adds the *TextBox* to the window. If the remaining space is smaller than the *TextBox*'s minimum size, there is no feasible solution. As ORCSolver has reached a leaf of the tree, if a feasible solution exists, the solver minimizes the quadratic objective function globally and stores the result as the currently best solution. Then the solver backtracks in the B&B tree to the *VerticalFlow* node and tries other alternatives that may lead to a better solution. It starts with the alternatives with fewer columns, *i.e.*, in the order of $numCols_{pref}$ - 1, $numCols_{pref}$ - 2, etc. We re-process the *VerticalFlow* with $numCols_{pref}$ - 1 columns first, and go down the tree to the *TextBox*. If we did not get a feasible solution in the previous branch, there may be a feasible solution in this branch, since, as we squashed the vertical flow area, there is now more space for the *TextBox*. Then, our currently best solution would be the leaf of this branch. If it was feasible in the previous branch, then the overall objective value may be smaller now. If it is smaller, we update the currently best solution accordingly and process the next *VerticalFlow* alternative with $numCols_{pref}$ - 2 columns. If that is not better than the the previous branch, ORCSolver stops trying even fewer columns since that would continue the trend of a narrower, taller *VerticalFlow* and lead to worse results. Analogously, it also tries increasing numbers of columns starting from $numCols_{pref}+1$ until that also increases the loss.

The overall idea of our approach is to first try the local best fit with the preferred number of rows/columns in the flow at each flow node and then gradually explore into the direction of decreasing overall loss by adding new branches to process for both smaller and larger numbers of rows/columns. After getting the best solution for the *VerticalFlow*, ORCSolver backtracks to process the *HorizontalFlow*. Similarly, our solver gradually explores in the directions of decreasing overall loss values. For each branch created for the *HorizontalFlow*, it proceeds down the branch following the same process for the *VerticalFlow* as in the main stem. By backtracking and exploring possible better solutions, it optimises the solution for the overall layout.

Constraint Solving

Each node in the B&B tree inherits all the variables, constraints and objectives from its parent and creates new boundary variables (top, bottom, left, right) for the corresponding sub-layout. If a sub-layout S has min/pref/max sizes, we attach the following constraints to it:

$$S.right - S.left = S.prefWidth\ [SOFT]$$
$$S.bottom - S.top = S.prefHeight\ [SOFT]$$
$$S.right - S.left \geq S.minWidth$$
$$S.bottom - S.top \geq S.minHeight$$
$$S.right - S.left \leq S.maxWidth$$
$$S.bottom - S.top \leq S.maxHeight$$

As a quadratic programming solver cannot handle soft constraints directly, and instead of adding the soft constraints to the system, we rewrite the two soft constraints by adding slack variables δ_1 and δ_2:

$$S.right - S.left + \delta_1 = S.pref_width$$
$$S.bottom - S.top + \delta_2 = S.pref_height$$

Similarly, we add slack variables to all other soft constraints. In addition, we add squares of all slack variables into the objective function, *i.e.*, with N soft constraints with slack variables δ_1, δ_2, ..., δ_N, we have the following objective function:

$$Minimize\ \ \delta_1^2 + \delta_2^2 + ... + \delta_N^2$$

ORCSolver runs the quadratic programming solver to solve the constraint system based on the variables, constraints, and objectives in a leaf node and get an objective value as the loss for the node to drive the B&B process.

IMPLEMENTATION

We implemented our solver in *Python* using a state-of-the-art quadratic programming solver, *OSQP* [4], as our default solver.

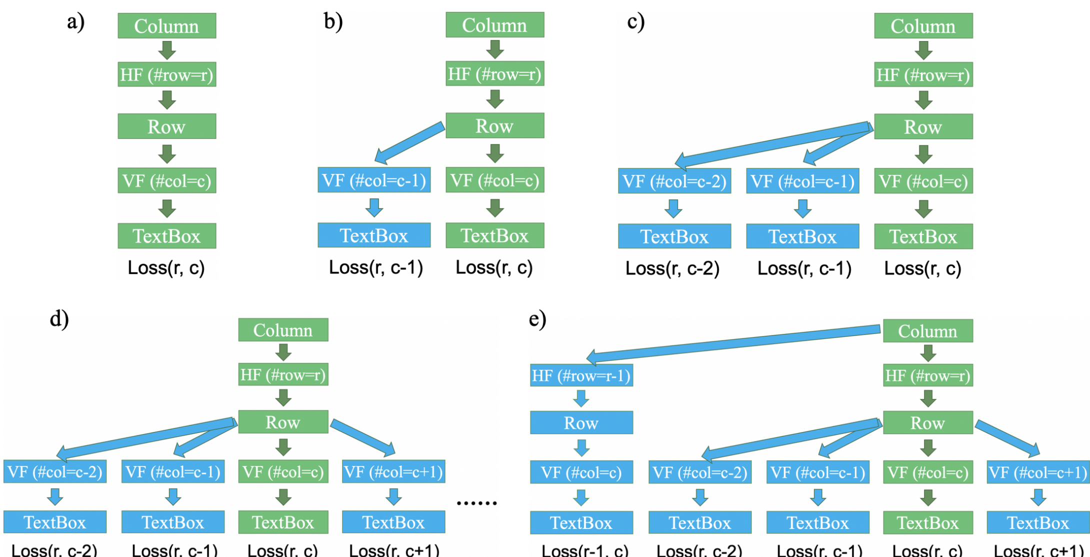


Figure 6. The solving process for the layout shown in Figure 3. After applying heuristics for flows, ORCSolver gets the preferred number of rows r in the horizontal flow and the preferred number of columns c in the vertical flow. a) ORCSolver constructs a tree for the layout based on the ORC notation and process it to get the penalty value for the main branch $Loss(r,c)$. b) ORCSolver adds a new branch for the vertical flow with $c-1$ columns and get its penalty value $Loss(r,c-1)$. If $Loss(r,c-1) \geq Loss(r,c)$, then it stops searching for smaller numbers of columns in the feasible range since the loss is already growing and smaller numbers can only lead to worse solutions. If $Loss(r,c-1) < Loss(r,c)$, ORCSolver stores this new branch as the currently best solution and c) move on to process smaller numbers of columns and repeat the same process. It stops trying smaller number of columns until a new branch is worse than its previous branch. d) ORCSolver tries larger numbers of columns in the same way. After finishing processing the vertical flow sub-layout, e) ORCSolver backtracks to the horizontal flow sub-layout and repeat the same process.

Solving with B&B is performed recursively with a method *solve*() that is implemented by all layout types, which makes it fairly easy to extend ORCSolver with new layout patterns and heuristics. ORCSolver with its API is available as open source from https://github.com/YueJiang-nj/ORCSolver-CHI2020.

EVALUATION

To compare ORCSolver and three other approaches to solve layout specifications with OR-constraints, we measured the runtimes required to solve ORC layouts. We conducted the experiments on a laptop with an Intel i5 CPU and measured the average execution time over 10 runs each with random feasible device sizes and random number of widgets in each flow, while keeping the total number of widgets constant. We found that the time varied only little between device sizes and was mainly determined by the total number of widgets. To avoid users touching complex constraints directly, Jiang et al. [38] proposed to describe layouts with ORC layout patterns. We based our work on the same approach and evaluated ORCSolver on six widely used layout patterns, including simple flow, connected flow, flow around a fixed area (all in Table 1), balanced flow, optional widgets, and alternative positions (all in Table 2). Most of these layouts are widely used in document layout and are not feasible with previous GUI layout systems.

The work on ORC layouts [38] used Z3 [10] to solve layout specifications with OR-constraints. Although various Mixed Integer Programming (MIP) and Satisfiability Modulo Theories (SMT) solvers exist for constraint-based systems, Z3 is the only solver that can deal with disjoint disjunctions and solve constraint systems with OR-constraints efficiently. Z3 dominates the competition in the SMT-COMP events [12] , as it is typically much faster than most competitors on most tasks. We compare ORCsolver with the following three approaches:

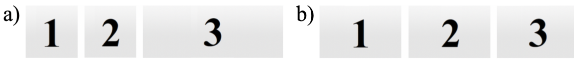


Figure 7. Layout results with a) Z3 Solver and b) our ORCSolver.

Pure Z3 Solver This is the original solving technique for ORC layouts [38] using the state-of-the-art Z3 solver for OR-constraint systems.

QP for Flows Each widget in flows has min/pref/max size constraints. This approach solves all these constraints in B&B nodes using the state-of-the-art OSQP quadratic programming solver, after applying our heuristics simplifying the flow constraints.

Pure B&B In addition to using B&B for optimising layout parameters, such as the number of rows/columns in flows, this approach also uses B&B to optimise the internal structure of the layouts, *e.g.*, solves flows with each branch representing different numbers of widgets in each row/column.

Our API allows us to plug in different solvers for different ORC patterns (including ORCSolver, “QP for Flows”, and “Pure B&B”). As shown in the logarithmic plots in Table 1 and Table 2, ORCSolver can solve layout patterns at near-interactive rates and about two orders of magnitude faster than other methods, especially when the number of widgets is large.

Differences in Outcomes

Different solving approaches may lead to different resulting layouts. Z3’s default *WMax* [55] optimization, which was

Figure 8. Different possible outcomes may have the same objective value.

used in the original ORC layout work [38], is fast but does not balance deviations between violated constraints, which leads to an unbalanced, less aesthetic appearance [75]. For example, as shown in Figure 7, if three widgets in a row have the same preferred size and the row width is not enough or too large for all of them to have their preferred size, then *WMax* [55] tries to make two of them have the preferred size to satisfy as many constraints as possible, which may lead to a bad layout result (Figure 7a). One can avoid this problem by applying a quadratic objective function within Z3, but this is prohibitively slow (several minutes for 10+ widgets) and none of Z3's other optimizers improves this. Many differences of Z3's outputs are due to its inability to solve quadratic functions tractably. By using quadratic programming, our solver distributes the deviations over the widgets and balances their appearance (Figure 7b). Thus, ORCSolver is not only faster than Z3, but also able to produce more balanced and aesthetic results. As complex layouts are often underspecified, several possible optimal solutions may exist. For example, as shown in Figure 8, placing the buttons either in the third row or a second column may have the same objective value, and different solvers may make different choices. The general challenge of underspecification can be addressed with suitable GUI editors [74, 77].

Computational Complexity

In the B&B process we start with a one-branch search tree and add more branches for different sub-layout parameters such as flows with different numbers of rows/columns. Yet, the number of branches we add is often very small as it is bounded by the feasible parameter range, *e.g.*, the minimum to maximum numbers of rows/columns in flows, and we stop adding branches in either direction (smaller or larger than the preferred number) once the previous branch has larger loss than the current best result. Hence, we only have an approximately constant factor of branches to add. The time complexity to process each branch from root to leaf is $O(numSublayouts)$. For each leaf, we use quadratic programming to solve the simplified layout specification, which has average polynomial complexity and is roughly linear in practice. Our heuristics for solving ORC patterns are $O(numWidgets)$. Therefore the overall complexity of ORCSolver is approximately linear in practice. By comparison, the original ORCLayout approach [38] used only Z3, which exhibits *exponential* runtime. Since "QP for Flows" solves a quadratic programming problem for each B&B node, its complexity is at least quadratic. "Pure B&B" is a brute-force method with exponential complexity.

DISCUSSION & CONCLUSION

In this paper, we presented ORCSolver, an efficient solver for adaptive GUI layout with OR-constraints. Our approach overcomes the main performance bottleneck for using OR-constrained layouts, and enables ORC systems to be solved efficiently at near-interactive rates even with large numbers of widgets. A formal notation for ORC layouts enables ORCSolver to support various types of layout patterns efficiently. Our modular approach of defining sub-layouts and corresponding solvers creates a framework that allows for further expansion, *e.g.*, for different layout patterns and solving strategies.

The novel ORCSolver approach is based on a branch&bound algorithm using interval arithmetic on layout parameters to limit the branching factor, so that OR-constraint systems for GUI layouts can be solved at near-interactive speeds. As illustrated through the performance graphs, our approach can solve various layout patterns much faster than previous work, typically by two orders of magnitude for larger layouts, which is a major improvement that becomes more important as GUI layouts get more sophisticated. The results show that it is now in principle feasible to solve ORC layouts on mobile devices.

Efficient solving of constraint-based layout systems also enables interactive modification and/or adaptation of layouts. Given that solving times reach at most 0.1 - 0.3 seconds, this means that ORCSolver can recompute layouts at speeds that are high enough to support a window resize or interactive GUI editing. Finally, our new approach enables designers to utilize the benefits of the unification of conventional constraint-based and flow layouts, which opens up new possibilities for GUI design. This is especially important for the definition of flexible GUI layouts that adapt seamlessly to screens with different sizes, orientations, or aspect ratios – while still being based on only a single layout specification.

Limitations and Future Work

ORCSolver solves a layout based on the formal specification of said layout. Thus, users currently have to specify a layout manually before feeding it into ORCSolver. Designers currently cannot visually specify ORC patterns through a GUI. Although there is a visual editor for ORC layouts, the editor does not yet allow users to specify resizing behaviour purely by direct manipulation. Inferring an ORC pattern from a layout is an avenue for future work. Additional heuristics such as meta-heuristics for optimization might further accelerate the solving process, but might also sacrifice accuracy. For example, large neighbourhood search [58] could find good candidate solutions by prioritizing promising search paths. We could also provide more heuristic rules and let the user choose which heuristic they want to use, or choose heuristics automatically based on runtime constraints, which can make our framework more flexible and generic. Finally, our current implementation does not support incremental solving. A warm-start solving option after a change in a layout is a useful functionality we plan to add in the future. Adding an incremental approach might reach real-time performance.

REFERENCES

[1] MOSEK ApS. 2017. *The MOSEK Optimization Toolbox for MATLAB Manual*. MATLAB Version 8.0 (Revision 57).

[2] 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. DOI: http://dx.doi.org/10.1007/354054742_40

[3] 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. DOI: http://dx.doi.org/10.1145/504704.504705

[4] Paul Goulart Alberto Bemporad Bartolomeo Stellato, Goran Banjac and Stephen Boyd. 2018. OSQP: An Operator Splitting Solver for Quadratic Programs. In *2018 UKACC 12th International Conference on Control (CONTROL)*. 339–339. DOI: http://dx.doi.org/10.1109/CONTROL.2018.8516834

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

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

[7] 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, 87–96. DOI: http://dx.doi.org/10.1145/263407.263518

[8] Bertrand Mercier Daniel Gabay. 1976. A Dual Algorithm for the Solution of Nonlinear Variational Problems via Finite Element Approximation. *Computers & Mathematics with Applications* (1976).

[9] George B. Dantzig. 1957. Discrete-Variable Extremum Problems. *Operations Research* (1957).

[10] 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, 337–340. DOI: http://dx.doi.org/10.1007/978-3-540-78800-_24

[11] Brian DeRenzi, Krzysztof Gajos, Tapan Parikh, Neal Lesh, Marc Mitchell, and Gaetano Borriello. 2008. Opportunities for Intelligent Interfaces Aiding Healthcare in Low-Income Countries. (01 2008).

[12] SMT-COMP Events. 2007. http://smtcomp.sourceforge.net/2007/participants.shtml. (2007).

[13] Hans Joachim Ferreau, Christian Kirches, Andreas Potschka, Hans Georg Bock, and Moritz Diehl. 2014. qpOASES: a Parametric Active-Set Algorithm for Quadratic Programming. *Mathematical Programming Computation* 6, 4 (01 Dec 2014), 327–363. DOI: http://dx.doi.org/10.1007/s12532-014-0071-1

[14] 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, 125–134. DOI: http://dx.doi.org/10.1145/1186562.1015789

[15] Marguerite Frank and Philip Wolfe. 1956. An algorithm for quadratic programming. *Naval Research Logistics Quarterly* 3, 1-2 (1956), 95–110. DOI: http://dx.doi.org/10.1002/nav.3800030109

[16] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. 1990. An Incremental Constraint Solver. *Commun. ACM* 33, 1 (Jan. 1990), 54–63. DOI: http://dx.doi.org/10.1145/76372.77531

[17] Krzysztof Gajos, David Christianson, Raphael Hoffmann, Tal Shaked, Kiera Henning, Jing Jing Long, and Daniel Weld. 2005. Fast and Robust Interface Generation for Ubiquitous Applications. 903–903. DOI: http://dx.doi.org/10.1007/11551201_3

[18] Krzysztof Gajos, Jing Jing Long, and Daniel Weld. 2006. Automatically generating custom user interfaces for users with physical disabilities. *Eighth International ACM SIGACCESS Conference on Computers and Accessibility, ASSETS 2006* 2006 (01 2006), 243–244. DOI:http://dx.doi.org/10.1145/1168987.1169036

[19] Krzysztof Gajos and Daniel Weld. 2005. Preference Elicitation for Interface Optimization. *UIST: Proceedings of the Annual ACM Symposium on User Interface Softaware and Technology* (01 2005), 173–182. DOI:http://dx.doi.org/10.1145/1095034.1095063

[20] Krzysztof Gajos and Daniel S. Weld. 2004. Automatically Generating User Interfaces For Ubiquitous Applications. In *Workshop on Ubiquitous Display Environments*.

[21] Krzysztof Gajos, Anthony Wu, and Daniel S Weld. 2005. Cross-device consistency in automatically generated user interfaces. In *Proceedings of the 2nd Workshop on Multi-User and Ubiquitous User Interfaces*. 7–8.

[22] Krzysztof Z. Gajos, Daniel S. Weld, and Jacob O. Wobbrock. 2008. Decision-Theoretic User Interface Generation. In *AAAI'08*. AAAI Press, 1532–1536.

[23] 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 (2010), 910–950. DOI: http://dx.doi.org/10.1016/j.artint.2010.05.005

[24] Krzysztof Z. Gajos, Jacob O. Wobbrock, and Daniel S. Weld. 2007. Automatically Generating User Interfaces Adapted to Users' Motor and Vision Capabilities. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology (UIST '07)*. ACM, 231–240. DOI: http://dx.doi.org/10.1145/1294211.1294253

[25] 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, 1257–1266. DOI: http://dx.doi.org/10.1145/1357054.1357250

[26] Michael Gertz and Stephen Wright. 2001. Object-Oriented Software for Quadratic Programming. *ACM Trans. Math. Software* 29 (11 2001). DOI: http://dx.doi.org/10.1145/641876.641880

[27] 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, 109–120. DOI: http://dx.doi.org/10.1145/168642.168653

[28] Eva Harb, Paul Kapellari, Steven Luong, and Norbert Spot. 2011. *Responsive Web Design*.

[29] 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, 117–124. DOI: http://dx.doi.org/10.1145/142621.142635

[30] 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, 218–232. DOI: http://dx.doi.org/10.1007/3-540-45349-_17

[31] Hiroshi Hosobe. 2011. A Simplex-Based Scalable Linear Constraint Solver for User Interface Applications. In *2011 IEEE 23rd International Conference on Tools with Artificial Intelligence*. 793–798. DOI: http://dx.doi.org/10.1109/ICTAI.2011.124

[32] Thibaud Hottelier and Ras Bodik. 2014. *Synthesis of Layout Engines from Relational Constraints*. Technical Report UCB/EECS-2014-181. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-181.html

[33] 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, 231–241. DOI: http://dx.doi.org/10.1145/2642918.2647378

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

[35] Gurobi Optimization Inc. 2016. *Gurobi Optimizer Reference Manual*.

[36] 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, 838–847. DOI: http://dx.doi.org/10.1145/1201775.882353

[37] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H.C. Yap. 1992. The CLP(R) Language and System. *ACM Transactions on Programming Languages and Systems* 14, 3 (5 1 1992), 339–395. DOI: http://dx.doi.org/10.1145/129393.129398

[38] Yue Jiang, Ruofei Du, Christof Lutteroth, and Wolfgang Stuerzlinger. 2019. ORC Layout: Adaptive GUI Layout with OR-Constraints. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM. DOI: http://dx.doi.org/10.1145/3290605.3300643

[39] 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, 137–153. DOI:http://dx.doi.org/10.1007/3-540-58601-_91

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

[41] 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., 53–60. DOI: http://dx.doi.org/10.1007/978-1-4842-2662-_20

[42] Christof Lutteroth and Gerald Weber. 2008. Modular Specification of GUI Layout Using Constraints. In *2008. ASWEC 2008. 19th Australian Conference on Software Engineering*. IEEE, 300–309. DOI: http://dx.doi.org/10.1109/ASWEC.2008.4483218

[43] Ethan Marcotte. 2017. *Responsive Web Design*. A Book Apart.

[44] Harry M. Markowitz and Alan S. Manne. 1957. On the Solution of Discrete Programming Problems. *Econometrica* 25 (1957), 84–110. http://dx.doi.org/10.2307/1907744

[45] Kim Marriott, Sitt Sen Chok, and Alan Finlay. 1998. A Tableau Based Constraint Solving Toolkit for Interactive Graphical Applications. In *Principles and Practice of Constraint Programming — CP98*, Michael Maher and Jean-Francois Puget (Eds.). Springer Berlin Heidelberg, 340–354.

[46] Kim Marriott, Peter Moulder, Peter J. Stuckey, and Alan Borning. 2001. Solving Disjunctive Constraints for Interactive Graphical Applications. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP '01)*. Springer-Verlag, 361–376. http://dl.acm.org/citation.cfm?id=647488.726811

[47] Kim Marriott and Sitt Sen Chok. 2002. QOCA: A Constraint Solving Toolkit for Interactive Graphical Applications. *Constraints* 7, 3 (01 Jul 2002), 229–254. DOI:http://dx.doi.org/10.1023/A:1020513316058

[48] Jacob Mattingley and Stephen Boyd. 2012. CVXGEN: a Code Generator for Embedded Convex Optimization. *Optimization and Engineering* 13, 1 (01 Mar 2012), 1–27. DOI: http://dx.doi.org/10.1007/s11081-011-9176-9

[49] 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. DOI: http://dx.doi.org/10.1177/154193120004400206

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

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

[52] 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.

[53] Yurii Nesterov and Arkadii Nemirovskii. 1994. *Interior-Point Polynomial Algorithms in Convex Programming*. Society for Industrial and Applied Mathematics. DOI: http://dx.doi.org/10.1137/1.9781611970791

[54] Arnold. Neumaier. 1991. *Interval Methods for Systems of Equations*. Cambridge University Press. DOI: http://dx.doi.org/10.1017/CBO9780511526473

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

[56] Brendan O'Donoghue, Eric Chu, Neal Parikh, and Stephen Boyd. 2016. Conic Optimization via Operator Splitting and Homogeneous Self-Dual Embedding. *Journal of Optimization Theory and Applications* 169, 3 (01 Jun 2016), 1042–1068. DOI: http://dx.doi.org/10.1007/s10957-016-0892-3

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

[58] Stefan Pisinger, David; Røpke. 2010. Large Neighborhood Search. *Handbook of Metaheuristics* (2010).

[59] Erica Sadun. 2013. *iOS Auto Layout Demystified*. Addison-Wesley Professional.

[60] 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, 275–284. DOI: http://dx.doi.org/10.1145/3197231.3197249

[61] Michael Sannella. 1994. Skyblue: A Multi-Way Local Propagation Constraint Solver for User Interface Construction. In *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology (UIST '94)*. ACM, 137–146. DOI: http://dx.doi.org/10.1145/192426.192485

[62] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. 1993. Multi-way versus one-way constraints in user interfaces: Experience with the deltablue algorithm. *Software: Practice and Experience* 23, 5 (1993), 529–566.

[63] 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. DOI:http://dx.doi.org/10.1016/j.infsof.2015.10.005

[64] 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, 167–177. DOI:http://dx.doi.org/10.1145/97924.97943

[65] 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, 544–554. DOI: http://dx.doi.org/10.1145/2786805.2786808

[66] Wolfgang Stuerzlinger, Olivier Chapuis, Dusty Phillips, and Nicolas Roussel. 2006. User Interface Façades: Towards Fully Adaptable User Interfaces. *UIST '06: ACM Symposium on User Interface Software and Technology* (10 2006). DOI: http://dx.doi.org/10.1145/1166253.1166301

[67] 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, 201–210. DOI: http://dx.doi.org/10.1145/120782.120804

[68] 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. DOI: http://dx.doi.org/10.1109/ASWEC.2010.38

[69] 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., 1613–1619. http://dl.acm.org/citation.cfm?id=1630659.1630944

[70] Philip Wolfe. 1960. The Simplex Method for Quadratic Programming. *Econometrica* 28, 1 (jan 1960), 170. DOI: http://dx.doi.org/10.2307/1905320

[71] 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, 27–34. DOI: http://dx.doi.org/10.1007/978-3-319-67744-_2

[72] 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, 465–466. DOI: http://dx.doi.org/10.1145/108844.109005

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

[74] Clemens Zeidler, Christof Lutteroth, Wolfgang Stuerzlinger, and Gerald Weber. 2013b. 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, 343–352. DOI: http://dx.doi.org/10.1145/2379256.2379287

[75] 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, 72–79. DOI: http://dx.doi.org/10.1145/2379256.2379268

[76] 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, 674–682. DOI: http://dx.doi.org/10.1145/2414536.2414638

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

[78] Clemens Zeidler, Gerald Weber, Wolfgang Stuerzlinger, and Christof Lutteroth. 2017b. Automatic Generation of User Interface Layouts for Alternative Screen Orientations. In *Human-Computer Interaction - INTERACT 2017*, Regina Bernhaupt, Girish Dalvi, Anirudha Joshi, Devanuj K. Balkrishan, Jacki O'Neill, and Marco Winckler (Eds.). Springer International Publishing, 13–35.