The Auckland Layout Editor: An Improved GUI Layout Specification Process

Clemens Zeidler, Christof Lutteroth
University of Auckland
Auckland 1010, New Zealand
{clemens,christof}@cs.auckland.ac.nz

Wolfgang Stuerzlinger
York University
Toronto, Canada M3J 1P3
wolfgang@cse.yorku.ca

Gerald Weber
University of Auckland
Auckland 1010, New Zealand
gerald@cs.auckland.ac.nz

ABSTRACT
Layout managers are used to control the placement of widgets in graphical user interfaces (GUIs). Constraint-based layout managers are among the most powerful. However, they are also more complex and their layouts are prone to problems such as over-constrained specifications and widget overlap. This poses challenges for GUI builder tools, which ideally should address these issues automatically.

We present a new GUI builder – the Auckland Layout Editor (ALE) – that addresses these challenges by enabling GUI designers to specify constraint-based layouts using simple, mouse-based operations. We give a detailed description of ALE's edit operations, which do not require direct constraint editing. ALE guarantees that all edit operations lead to sound specifications, ensuring solvable and non-overlapping layouts. To achieve that, we present a new algorithm that automatically generates the constraints necessary to keep a layout non-overlapping. Furthermore, we discuss how our innovations can be combined with manual constraint editing in a sound way. Finally, to aid designers in creating layouts with good resize behavior, we propose a novel automatic layout preview. This displays the layout at its minimum and in an enlarged size, which allows visualizing potential resize issues directly. All these features permit GUI developers to focus more on the overall UI design.

ACM Classification Keywords
H.5.2 User Interfaces: Graphical user interfaces (GUI)

Author Keywords
GUI builder; layout editing; constraint-based layout; layout manager

INTRODUCTION
The use of graphical user interfaces (GUIs) is widespread, including on web and mobile platforms. WYSIWYG GUI builders facilitate the creation of GUIs by designers. In such tools, widgets are dragged and dropped from a palette onto a GUI canvas. Subsequently, they are selected, moved, resized, and adjusted in order to compose the final GUI layout.

Nowadays the same application may be used across a wide range of screen sizes, such as desktop machines, tablets and mobile phones. Resizable GUIs are also recommended by modern GUI guidelines[1]. Consequently, it is important that a GUI stays sound at different sizes. To create resizable GUIs with dynamic layouts, a *layout manager* can be used. A layout manager implements a *layout model* that defines how objects in a layout, the *widgets*, can be arranged and how their resize behavior can be specified.

Constraint-based layout models are naturally powerful: with the notable exception of flow layouts, many other layout models, including gridbag layout [26], can be reduced to constraint-based layouts (see also Figure 1). For example, the "LCD" display in iTunes[2] is always centered at the top of the window between other widgets, but this is usually not possible with other layout models. Many other layout managers rely on a hierarchy of nested layouts to define more complex layouts. Within nested layouts, widgets typically cannot be aligned across different levels of the hierarchy. In contrast, constraint-based layout models greatly reduce the need for nested layouts, even for visually hierarchical layouts. Constraints can align widgets that are situated in different parts of a visually hierarchical layout [17]. Constraint-based layout managers use a dedicated or off-the-shelf constraint solver to compute widget positions and sizes.

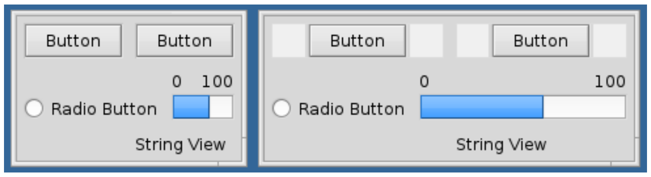


Figure 1. Example constraint-based layout. The behavior of the middle row is independent from the top and bottom rows when resizing. This is impossible to achieve with a single gridbag layout.

While constraint-based layouts are more powerful, their creation may be more complex and poses challenges. General constraints are difficult to visualize and even harder to manipulate directly. Specifying individual constraints can be tedious and error-prone, as they are situated at a low level of abstraction. Thus widgets may overlap each other (Figure 2)

[1]`developer.android.com/training/multiscreen/screensizes.html`
`msdn.microsoft.com/en-us/library/windows/desktop/aa511331.aspx`
[2]`apple.com/itunes`

or become over-constrained, i.e., the layout may have no solution. This makes it harder for GUI designers to leverage the advantages of constraints.

The novel Auckland Layout Editor (ALE) addresses these issues and simplifies the creation and modification of constraint-based layouts. Here, we present a complete description of ALE's layout edit operations, which maintain constraints automatically and hence require fewer operations to configure widgets correctly. In particular, we present a new method for automatically identifying all constraints necessary to maintain a given layout, based on a tiling of orthogonal polygons into rectangles, and we explain how this method is used during layout editing. The output of this method ensures that layouts stay non-overlapping for all sizes and that they always have a unique solution. While ALE's edit operations enable quick creation of the most common layouts, arbitrary constraints can also be added via auxiliary dialogs. We demonstrate how ALE's edit operations can work together with general constraint editing while keeping the layout specifications sound, i.e., non-overlapping and solvable. A recent user study found that ALE's edit operations can make layout creation and editing significantly faster than other modern GUI builders [29]. Furthermore, we present a novel approach to visualizing the resize behavior of a window. Layout design usually happens at a fixed size, yet the resulting layout should also look good at different sizes. Currently this requires explicit manual testing by the designer. To help the designer to better evaluate the resize behavior, we present a new preview that automatically shows appropriately shrunk and enlarged versions during editing.

Contributions

We present several innovations and improvements to the GUI layout creation process. In particular, we introduce:

- Detailed specifications of edit operations for constraint-based layouts that automatically keep a layout sound and solvable and speed up layout editing.
- A new algorithm that automatically adds constraints at design-time to prevent widget overlap at all sizes.
- An extension to permit manual constraint addition and editing in the system.
- A novel way to visualize the resize behavior of a layout during the design phase.

REQUIREMENTS FOR LAYOUT EDITING

In this section we specify our design goals for ALE. These are derived from related work, existing usability guidelines, and our own experience with layout creation. We also define requirements for all layout specifications that can be created with a GUI builder. We differentiate between concrete layouts, as they are rendered on a screen, and *layout specifications*, which can be rendered at many different sizes and hence lead to many concrete layouts. Our design goals can be subdivided into usability, completeness, and soundness requirements. Common usability guidelines [14, 18] naturally apply to GUI builders. In particular, a builder should be intuitive, easy to use, and edit operations should lead to predictable results. Common GUI design guidelines, such as [6], emphasize that it is not desirable to place widgets completely "freely", i.e., at an arbitrary position in a layout. Aligned layouts are more compact and easier to understand [9]. Consequently, widgets should be easy to align and the system should automatically maintain such alignments.

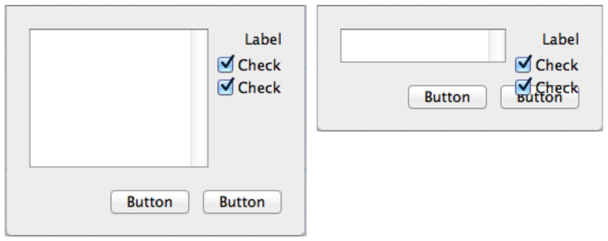


Figure 2. The constraint-based layout on the left appears sound. Yet, at the smaller size shown on the right, widgets overlap due to a missing constraint. The constraint-based GUI builder used, Apple Xcode, gave no indication of this issue.

Completeness guarantees that every correct layout specification that is both possible in the underlying layout model and appropriate in practice can actually be created.

Soundness ensures that every layout specification that can be created is correct and adequate, i.e., no erroneous layouts can be created. Here we use three soundness requirements.

Unique Solution A layout specification must have exactly one solution. When computing a layout, the underlying specification needs to be *solved*, i.e., a concrete visual layout needs to be generated. If the specification contains conflicts, then there is no solution. If there is more than one solution, then the layout may behave non-deterministically. For example, if the position of a widget is underspecified it may "randomly" change its position during resizing, which is undesirable.

Non-Overlap A layout specification must be non-overlapping, i.e., in all layouts derived from the specification, widgets may not intersect each other or the boundaries. This property ensures that all widgets are completely visible and accessible at all GUI sizes. A widget that is overlapped by another may be inaccessible and thus useless (Figure 2).

Well-Defined Layout Sizes Each layout specification must have a single minimum, preferred, and maximum layout size. It is theoretically possible for a layout specification to have multiple valid extreme sizes (Figure 3). Yet, such situations are not desirable. First, window managers support only a single minimum and maximum window size. Secondly, in order to keep the layout non-overlapping, "or" constraints must be used, which are difficult to solve. Last, a layout with multiple extreme sizes potentially has a very different appearance at different sizes, which may confuse users. Note that we still permit discrete layout specification changes at runtime. For example, when the screen is rotated one can switch between different layout specifications with different extreme sizes.

RELATED WORK

Myers [19] summarized various non-constraint-based techniques for creating GUIs, including graphical tools for placing interface objects on screens. IBuild [25] enabled the cre-

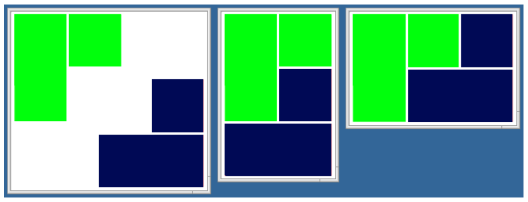


Figure 3. Layout with two possible minimum sizes.

ation of complex GUIs in a WYSIWYG manner. It already supported nested layouts, spring-like layout elements, and interactive testing of resizing. Druid [23] predicted intended alignment and spacing of a widget during editing, facilitating placement. FormsVBT [1] supported simultaneous editing of a textual and a graphical representation of a layout.

Some GUI builders permit manual constraint editing. OPUS used tabstops for specifying constraints [10]. Lapidary [27] provided rich constraint editing functions. In the Gilt system [8], widgets could be aligned relative to user-specified tabstops or other widgets. In the Intui GUI builder [22], constraints can be toggled between struts and springs, between and inside widgets. Yet, only one constraint per widget side is supported, which can make it impossible to create non-overlapping layout specifications. Peridot [20] analyzes objects drawn by the user and guesses constraints to be added. The user then has to confirm the proposed constraints. Rockit [16] automatically proposes constraints based on the "gravity field" of other objects. The user can select the desired constraints from a set. This is similar to previous work where constraints are inferred by snapping graphical objects relative to other objects [11]. In ALE widgets can be snapped to others in order to set up the corresponding constraints. However, ALE adds more powerful edit operations, such as inserting a widget between others and swapping widgets. Bramble [7] connects objects using a set of interactors, which establish non-linear constraints. While this can be used to prevent overlap, the user has to add interactors manually. ALE adds such constraints automatically.

Today, there are many open-source GUI builders, such as WindowBuilder Pro[3], Matisse/Swing[4], and Qt Designer[5]. There are also commercial GUI builders, such as MS Expression Blend[6] and Visual Studio[7]. Most of them support aligning widgets via snapping, but do not maintain alignment when objects are moved or resized. Also, layouts must be resized manually in order to evaluate their resize behavior. In contrast, ALE's edit operations keep widgets aligned by default. ALE also provides an automatic preview of resize behavior.

Apple added support for constraints to the Xcode Interface Builder[8] in 2012. It aligns objects via snapping, but permits also free placement of widgets. Constraints between widgets can be added manually but are removed upon conflict. All the abovementioned constraint-based approaches require that the user knows how to specify constraints manually. In contrast, ALE enables the creation of many interesting layouts without exposing a single constraint to the user. Moreover, the edit operations automatically maintain all constraints for snapped items and for non-overlap. ALE disables conflicting manual constraints to support error diagnosis. Finally, a recent comparison between Xcode and ALE confirms that our new approach results in better usability [29].

Adaptive document layout [12] in the print and web domains is somewhat similar to GUI layout. However, the flow of a document constrains placement differently than a GUI. Documents typically arrange text and images in a sequential manner, and support more flexibility in the layout using algorithms for figure placement, line-break, and pagination. Grid-based [15] as well as constraint-based [3, 13] methods have been used for document layout.

Firefox's responsive design view[9] can simulate a GUI at different screen sizes. Also Android Studio[10] shows layouts at different device sizes. However, this does not take an optimally enlarged layout size into account. FormsVBT [1] supports a graphical and a textual edit view as well as a result view. However, the result and graphical edit views are displayed at a fixed size and give no indication of how a layout would look at a different size. In contrast, ALE shows a live preview of the layout at the minimum and an optimally enlarged layout size. This reduces the potential for user errors, as the resize behavior is visualized immediately and directly. ALE's layout previews are targeted particularly at situations where enough constraints exist, but do not lead to the desired resize behavior. In these situations, displaying all constraints clutters the design view, which does little to help the designer quickly identify the problem.

THE AUCKLAND LAYOUT EDITOR (ALE)

Similarly to other GUI builders, ALE uses a component palette and an editing canvas (Figure 4). Widgets are dragged from the palette into the editing canvas, where the designer can change the layout using a rich set of edit operations. We first describe the constraint system used in ALE, which is conceptually similar to other such systems.

A layout consists of various types of *layout items*, such as widgets and nested layouts. In the following and for brevity, we refer to all layout items simply as widgets. Each widget has an intrinsic *minimum*, *preferred*, and *maximum size*. A widget assumes its preferred size if there are no other constraints for it, similarly to the behavior of a sponge. ALE includes a special type of widget that can act as a strut or spring.

Layout items are connected via constraints. ALE supports two types of linear constraints: *hard* constraints, which have

[3]`eclipse.org/windowbuilder`
[4]`netbeans.org/features/java/swing.html`
[5]`qt.nokia.com`
[6]`microsoft.com/expression/products/Blend_Overview.aspx`
[7]`microsoft.com/visualstudio/en-us`
[8]`developer.apple.com/technologies/tools`
[9]`developer.mozilla.org/en-US/docs/Tools/Responsive_Design_View`
[10]`developer.android.com/sdk/installing/studio.html`

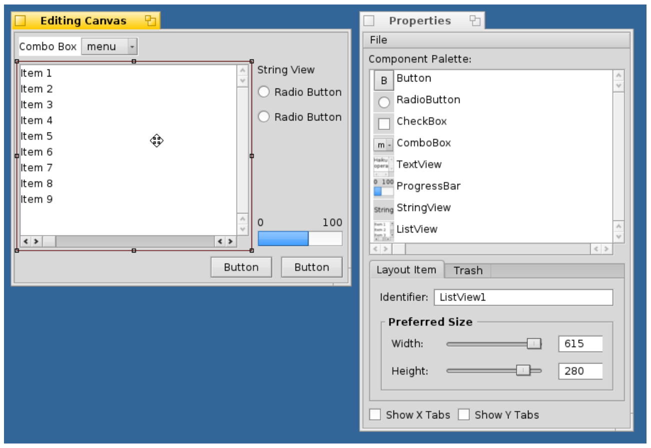


Figure 4. Screenshot of the ALE GUI builder. New widgets can be inserted into the editing canvas on the left via drag and drop from the component palette on the right.

to be satisfied, and *soft* ones, which may be violated if necessary. We support both equality and inequality constraints.

Variables in a constraint are called *tabstops* and represent horizontal or vertical grid lines. Other frequently used names for the same concept are aligners, guides, and snap or anchor lines. Each window defines tabstops for its four borders, so that they can be used for alignment. Tabstops are reference counted, which allows unused ones to be removed automatically.

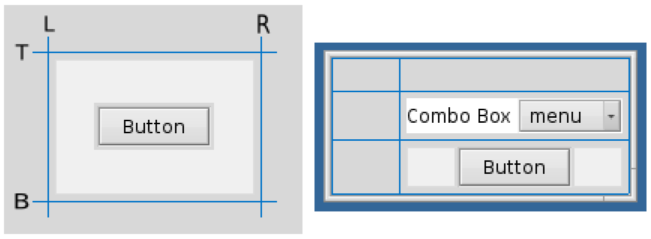


Figure 5. Left: Widgets are by default automatically centered in their layout area (light gray). The area is surrounded by a margin and four tabstops. ALE shows tabstops as blue lines. Right: A combo box and a button are connected to the same vertical tabstops. As the combo box cannot shrink further, the button is centered.

Within the overall layout, each widget is associated with a rectangular *layout area* that is surrounded by four tabstops. As most layouts use margins, there is by default a small margin between the layout areas and their tabstops (Figure 5). When inserting a widget into a layout, some constraints are automatically derived from the intrinsic sizes of a widget: a hard inequality for the minimum, a soft equality for the preferred, and a soft inequality for the maximum size. These constraints are defined both horizontally and vertically and are managed by the associated layout area. Consequently, users do not need to manage constraints for intrinsic sizes manually. The preferred size of a widget can be fine-tuned in the properties window. This is comparable to changing the "weight" of a row or column in a gridbag layout. If the size of the layout area is between a widget's minimum and maximum size, the widget uses the whole space of the area. If the widget cannot grow, it is centered in the area by default (Figure 5). The user has the option to align the widget to any border or corner of the layout area. Also, the user can fine-tune widget positions by changing the margins.

In contrast to most other systems, ALE uses a quadratic optimization function with an active set solver [5]. This method is fast enough for GUI problems [4]. We found that solving layouts with 30 widgets can be done in the order of 10 ms (Intel Core 2 Duo 3GHz). Soft preferred size constraints together with a quadratic optimization function ensure that there is a unique layout solution, as a convex objective function has a unique minimum. A linear optimization function cannot ensure this, as there is in general an infinite number of solutions for the soft constraints, which leads to underdetermined widget sizes [28].

Many useful layouts can be specified simply by aligning layout areas with each other. Such layouts naturally reuse tabstops for multiple layout areas and thus need no additional constraints. See, e.g., Figure 5, where the top tabstop of the button is reused as the bottom tabstop of the combo box. A layout consisting of fully connected layout areas is always solvable as long as the containing window is large enough, because the maximum constraints are soft and thus no conflicts can occur (e.g., in Figure 5 the button's maximum size constraint is violated, as it has a fixed size).

ALE LAYOUT EDIT OPERATIONS

Ideally, a GUI builder should make the creation of a constraint-based layout easy and intuitive. If a GUI builder exposes each constraint, margin, tabstop, strut or spring directly to the user, this significantly increases the complexity of layout editing. In other words, the user has to create, correctly configure, and maintain a substantially larger number of entities to achieve a given result. To avoid this, ALE's layout edit operations were designed to automate the construction and maintenance of the layout specification as much as possible, while still ensuring that the operations are sufficient to create and edit complex GUIs.

As in most GUI builders, a GUI can be created and edited by drag and drop operations. The edit operations provided are moving, swapping, resizing, inserting, and removing of a layout area and the widget contained therein. All operations try to connect layout areas to existing tabstops through snapping, and create as few new tabstops as possible. Reusing tabstops leads to good alignment and simpler layout specifications that are easier to understand. Only the intrinsic size constraints are updated when a layout area is connected to new tabstops, and new constraints are added only in a few cases. Furthermore, all operations are designed to leave an initially sound layout in a sound state. Thus no overlap between widgets can be generated. In the following sections and for brevity, we describe only the horizontal case for each operation. Vertical cases are handled analogously.

Edit operations are started by dragging a widget or a border of a selected widget. Then, ALE checks at each mouse position whether an operation can be applied and gives corresponding visual feedback (Figure 6). This is done by applying the operation tentatively in the background and checking the result. If an operation is applicable, the user can commit it by "dropping" the widget, and the result becomes visible. In general, edit operations can affect multiple widgets and a layout may differ in several places after an operation. To help the user understand the changes, a short animation visualizes how affected widgets are altered. All edit operations can be reverted using undo. Inserting, removing, moving and swapping of widgets are done via dragging the widget; dragging a border is used for resizing.

Inserting a Widget
New widgets can be inserted into the layout by drag and drop from the widget palette into the edit canvas. When dragging a widget, it is visualized as a dotted rectangle, the *dragged widget outline*, which has the preferred size of the dragged widget. We limit the size of the dragged widget outline to a reasonable one to avoid problems with very large widgets. Two cases need to be considered when inserting a new widget into the layout. A widget can be inserted into an empty area (Figure 7) or it can be placed between an existing tabstop and a widget adjacent to that tabstop (Figure 8).

Inserting into an Empty Area
A widget can be inserted into an empty area if the mouse position is in that empty area. Dropping the dragged widget into an empty area must constrain that widget to at least two tabstops, one in each direction, because of the requirement for a unique solution. If an edge of the widget is close to a tabstop, the widget is snapped to that tabstop. If the widget can only be snapped to a single tabstop in one direction, a new tabstop is created at the opposite widget edge.

If the widget cannot be snapped to any existing tabstop in a given direction, we solve the problem in a novel way. In this case, the largest rectangular empty area that completely contains the dragged widget outline is identified (Figure 6). If the outline overlaps another non-empty layout area, i.e., no rectangular empty area completely contains the outline, the largest empty rectangular area that contains the mouse position is used. For the direction in question, the widget is then connected to the outer tabstops of this largest rectangular empty area. This is also an easy way to fill a whole rectangular empty area with a resizable widget.

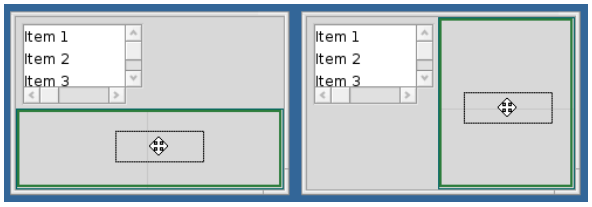


Figure 6. ALE automatically selects the largest empty area, depending on user input. The area where the widget will be inserted when dropped is highlighted in green. Here, the widget is entirely within the lower empty area (left) or within the right one (right). In the lower-right quadrant, the larger (right) area would be used.

These rules enable the user to select the desired area in an intuitive way. The combination of being able to use the full extent of an empty area or simply to snap a widget to one corner or border makes this operation quite versatile (Figure 7), see also the Tk pack model[11].

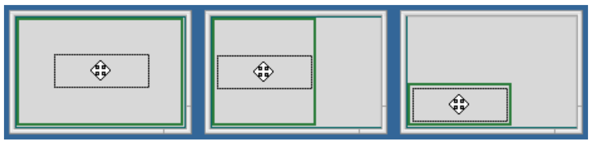


Figure 7. A widget can be inserted at various positions within an empty rectangular area. Left: Far from the tabstops, the dragged widget fills the whole empty area. Middle: When close to only the left tabstop, the widget snaps to that tabstop and resizes vertically. The right edge creates a new tabstop. Right: When close to the left and bottom tabstops, the widget is constrained to the corner.

Inserting between Tabstop and Area
A widget can be inserted at any existing tabstop. To trigger this operation, the mouse position must be over an existing widget and close to a tabstop, i.e., an edge, of that widget. One edge of the new widget then aligns with that existing tabstop, while a new tabstop is created for the opposite edge. The existing widget is connected to the new tabstop (Figure 8). Initially, the size of the new widget is *temporarily* set to zero, as the final size is determined later by the solver.

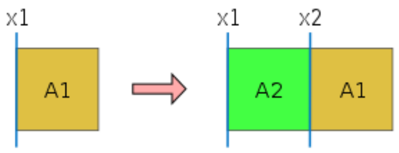


Figure 8. When inserting a new widget $A2$ between area $A1$ and tabstop $x1$, a new tabstop $x2$ is added.

Once the new widget has been inserted at an existing tabstop in one direction (say horizontal), it must also be connected in the other (vertical) direction. If its preferred height is smaller than the height of the existing widget, the new widget is either connected to the top or the bottom tabstop of the existing widget, whichever is closer. In this case, a new tabstop is added at the opposite vertical side of the new widget. Figure 9 shows such an example where a small button is inserted beside a larger text view. If the preferred height of the new widget is larger or close to the height of the existing widget, the new widget is connected to both the top and the bottom tabstop of the existing widget. Because the maximum size constraints of the existing widget are soft, this never generates a conflict.

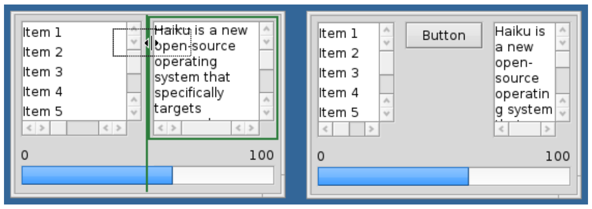

Item 1
Item 2
Item 3
Item 4
Item 5
Haiku is a new open-source operating system that specifically targets
0
100
Item 1
Item 2
Item 3
Item 4
Item 5
Button
Haiku is a new open-source operatin g system
0
100

Figure 9. Inserting a small button at an existing tabstop between a list and a text view. The existing tabstop and widget are highlighted in green. As the button is dropped close to the top of the text view, it is connected to its top tabstop.

[11]`tcl.tk/man/tcl8.6/TkCmd/pack.htm`

Removing Widgets
A widget is removed from the layout by dragging it outside the edit canvas. Any created gaps are filled, see below.

Moving Widgets
When moving a widget, a valid position for the insertion is determined and then the widget is moved from its original place. Here the same logic is used as for insertion. However, when looking for an insertion position the area occupied by the moving widget is ignored. This makes it impossible to snap a widget to itself, for example.

Swapping Two Widgets
Dropping one widget onto another swaps the positions of the two widgets. Here it is sufficient to connect the moved widget to the tabstops of the other one and vice versa.

Resizing a Widget
When a widget is connected to wrong/undesired tabstops, the user can adjust this with a resize operation. Resizing is done by dragging one of the borders or corners of a widget. During resizing, all relevant tabstops are visualized as light blue lines to aid alignment. There are two cases to consider. First, a widget may be resized to an existing tabstop, via snapping (Figure 10). Here, the system ensures that the resized widget does not overlap with any other one; otherwise, the resize operation is not permitted. Secondly, a widget can be resized to match its preferred size. This occurs when the dragged border is released in an empty area, i.e., without snapping. In this case, a new tabstop is created for the dragged border (two for a corner). This can also be interpreted as detaching a widget from a tabstop. Preferred sizes can be adjusted via the properties window (Figure 4 bottom right).

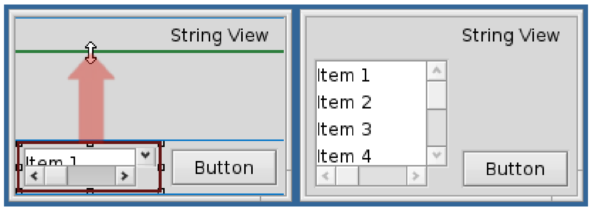


Figure 10. Resizing the top of the list widget to the bottom of the string.

Filling Gaps
A move, resize, or remove operation can detach adjacent widgets as one or more widgets may lose their connection to a tapstop. This is usually visible as a "gap". Such "gaps" and the associated "floating" widgets violate the unique solution requirement (Figure 11). ALE avoids this by checking for unconnected widgets after remove, move and resize operations. If the layout contains parts that are unconnected, all "floating" widget groups are moved one after another into the direction where the removed widget has been located. For example, if a group was connected to the right side of the removed or resized widget, the group is moved to the left. When the foremost widget of the floating group hits the border of another widget, it is connected with the corresponding tabstop of the other widget (Figure 11). During this process the moved group may become connected to another floating group. Groups are moved until they are connected directly or indirectly to at least one horizontal and one vertical layout border.

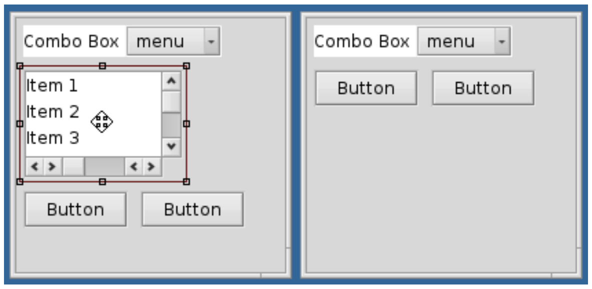


Figure 11. Filling gaps: Upon removal of the list widget, the two buttons both "float" vertically. Moving the button group up fills the gap. The tops of the buttons are connected to the bottom of the combo box.

NON-OVERLAPPING LAYOUTS
We start with a formal definition of *non-overlap* to explain how ALE guarantees non-overlapping layouts. A widget is *completely left* of another if the right side of the first widget is left of the left side of the other widget. Two widgets are called *horizontally non-overlapping* if either one is completely left of the other. *Vertically non-overlapping* is defined analogously. Two widgets are *non-overlapping* if they are horizontally or vertically non-overlapping. A layout is non-overlapping if all possible pairs of widgets are non-overlapping.

While a given layout may be non-overlapping, this does not imply that the underlying specification produces only non-overlapping layouts. This is a central problem in GUI layout. For example, the layout on the left of Figure 2 is non-overlapping. Yet, as the size is reduced, the check box starts to overlap the button due to a missing constraint. Any specification that produces only non-overlapping layouts is a *non-overlapping specification*.

The main idea of the non-overlap algorithm presented here is that for a given non-overlapping layout, the underlying specification can be made non-overlapping by adding additional *non-overlap constraints*. These are simple hard linear constraints that ensure a non-negative distance between two widgets. In the following, we show that all layouts created using ALE's operations are non-overlapping. Moreover, we explain where non-overlap constraints are added.

Non-Overlap of Created Layouts
All edit operations transform a non-overlapping layout so that it stays non-overlapping. Starting from the naturally non-overlapping empty layout we prove via structural induction that all layouts evolving from there are non-overlapping. Inserting or moving a widget into an empty area keeps the layout non-overlapping by definition. When inserting or moving a widget horizontally between a tabstop and another widget, it does not overlap the involved widget as a new "column" is created between the tabstop and the existing widget, and the new widget is temporarily assigned zero size. Furthermore, the new widget does not overlap any existing widgets above or below in a vertical direction because per definition

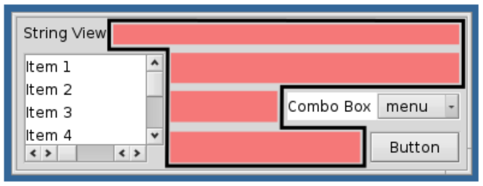


Figure 12. The empty area is an orthogonal polygon (black outline) that is filled with tiles (red). Each tile has a minimum size of zero, which guarantees a non-overlapping layout specification.

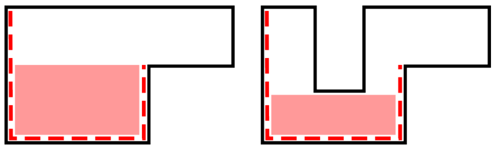


Figure 13. Empty polygon (black outline). An identified U-shape is visualized as a red dotted line. Left: The U-shape is tiled up to the shorter leg (red tile). Right: A widget extends into the U-shape and the U-shape is tiled only partially.

of the operation it has at most the height of the existing widget, and this widget did not overlap any other widgets before. Swapping two widgets does not change the topology of the tabstops and thus works as well. When increasing the size of a widget, the system always verifies that the resized widget does not intersect with another; otherwise the operation fails. Decreasing a widget's size also keeps the layout non-overlapping because the resulting widget is always smaller than the old one.

The last case that has to be considered is gap filling. When filling a gap, floating groups are moved one after another into the direction of the removed widget. Because a floating group is only moved until it hits another widget, this operation also keeps the layout non-overlapping. Given that each edit operation leaves the layout non-overlapping, the system can never get into a state where the layout has overlaps. Thus ALE is suitable for creating and editing a concrete layout under the non-overlap requirement. In the next section we discuss how we expand this to keep all layout specifications non-overlapping.

Non-Overlapping Layout Specifications

To maintain the non-overlap requirement of a layout specification during all resize operations, one needs to ensure that all pairs of widgets are non-overlapping. Yet, checking all pairs would be inefficient, so we propose a more sophisticated approach. We first recall that in *full layouts*, i.e., layouts without empty space, all widgets share tabstops with all neighbors. Thus the layout is non-overlapping.

Tiling of Empty Areas

It remains to argue what to do if a layout has one or more empty area(s). If two visually adjacent widgets do not share a tabstop, the gap between the two widgets is treated as an empty area with no extent. Otherwise, the empty area(s) are orthogonal polygons. Our solution is to tile these orthogonal polygons by introducing new, virtual, empty rectangular widgets (Figure 12). Each such virtual rectangular widget is connected to existing tabstops. These virtual widgets only need a minimum size of zero to guarantee non-overlapping layout specifications.

We now discuss the algorithm for tiling all empty areas. All empty areas are orthogonal polygons. In general, such polygons can be tiled in $O(n^{3/2} \log n)$ time [24] (n being the number of polygon edges), producing $O(n)$ tiles. As discussed later, the algorithm presented here permits a choice of the shape of the inserted tiles, which affects the resizing behavior of the layout. In a first step, all U-shaped segments in the orthogonal polygons are identified. A U-shape has a horizontal edge and two upwards-pointing vertical legs, the two neighboring edges. In general, the length of the legs can differ (Figure 13). In a second step, the identified U-shapes are tiled. Each U-shape is tiled up to its shorter leg (Figure 13 left). As depicted on the right of Figure 13, it is possible that a widget may extend into a U-shape. In this case the U-shape is only filled up to this widget. When inserting a new tile, new U-shapes may be created and these U-shapes must be added to the list of U-shapes.

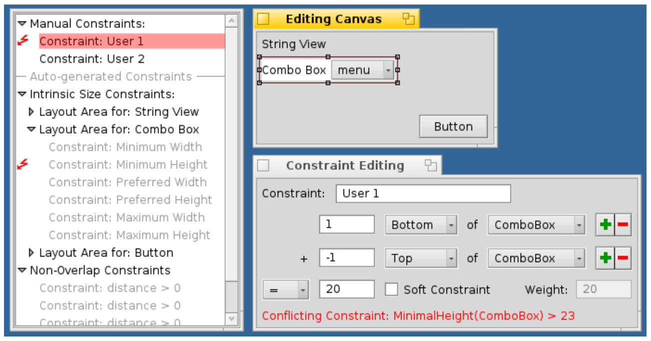
After tiling all U-shapes, all orthogonal polygons are completely tiled and the layout becomes non-overlapping. Moreover, the requirement that a layout has well-defined layout sizes is satisfied, as the inserted tiles conserve the topology of the widgets relative to each other. When tiling U-shapes, the inserted tiles become “row-like” (Figures 12 and 13). By choosing C-shapes instead, tiles become “column-like”. This may affect the resize behavior, i.e., if widgets move horizontally or vertically relative to each other. By default, ALE produces “row” tiles, but the user can toggle this.

If there are n widgets in the layout, then the empty areas cannot have more than $4n + 4$ corners. This gives us a small linear bound on the number of necessary constraints, as we need only two minimum size constraints per tile. In general, the tiling will have to be recomputed if a layout is modified.

After adding all non-overlap constraints identified by this algorithm the layout becomes non-overlapping. Moreover, the requirement that a layout can only have a single minimum and preferred size is satisfied, as the inserted tiles keep the orientation of the widgets towards each other constant.

When to Add Non-Overlap Constraints

For all edit operations discussed above and to test an operation for applicability, the operation is temporarily applied, the resulting specification solved, and then it is verified for soundness. As a first step, all existing non-overlap constraints are removed from the current layout so as not to interfere with the change. After the layout operation is temporarily applied, non-overlap constraints are inserted for the new specification. If the resulting specification is solvable, the operation can be applied. Otherwise the previous layout and non-overlap constraints are restored. As a last step of a successful operation, the solver recalculates the minimum, maximum and preferred layout sizes. If necessary, the parent window size is updated to fit the modified layout.


Manual Constraints:
Constraint: User 1
Constraint: User 2
Auto-generated Constraints
Intrinsic Size Constraints:
Layout Area for: String View
Layout Area for: Combo Box
Constraint: Minimum Width
Constraint: Minimum Height
Constraint: Preferred Width
Constraint: Preferred Height
Constraint: Maximum Width
Constraint: Maximum Height
Layout Area for: Button
Non-Overlap Constraints
Constraint: distance > 0
Constraint: distance > 0
Constraint: distance > 0
Editing Canvas
String View
Combo Box
menu
Button
Constraint Editing
Soft Constraint
Weight:

Constraint: User 1

1 Bottom of ComboBox

+ -1 Top of ComboBox

= 20 Soft Constraint Weight: 20

Conflicting Constraint: MinimalHeight(ComboBox) > 23

Figure 14. Left: Constraint list. Top-right: Editing canvas. Bottom-right: Constraint edit dialog. The selected constraint "User 1" ($ComboBox_{height} = 20$) conflicts with the minimum height constraint of the combo box ($ComboBox_{height} \geq 23$). Conflicting constraints are marked in the list and the corresponding widget is highlighted.

GENERAL CONSTRAINT EDITING

Using the new edit operations described above it is already possible to specify a large set of layouts. However, it may be necessary to add additional constraints manually in order to achieve certain layout specifications. For example, the width of a widget may be specified to be a multiple of the width of another. Another case is placing a widget globally centered in a layout, between other widgets. To fulfill the completeness requirement, such general constraint editing has to be supported. This section describes how manual editing techniques for linear hard and soft constraints can work together with the new edit operations. Moreover, we discuss how a layout can still be kept sound (non-overlapping and solvable).

The first step towards general constraint support is to permit the creation of arbitrary horizontal or vertical tabstops, not associated with widgets. Adding new tabstops relative or absolute to other tabstops in a constraint-based layout has been proposed [8]. In existing constraint-based GUI builders, constraints can be specified between two tabstops of different widgets. For general constraint editing, one only has to extend this to support arbitrary tabstops and also a (theoretically) unlimited number of tabstops. It is still possible to use the new edit operations introduced above, as they do not rely on every tabstop being connected to a widget. However, we still have to discuss how a layout specification can be kept non-overlapping. Furthermore, the creation or modification of manual constraints may create conflicts with other constraints. Figure 14 shows ALE's constraint editing interface with an example. Conflicting manual constraints are marked in the constraint list.

Ensuring Non-Overlapping Layouts

With our new edit operations it is only possible to connect widgets to others in the layout. Thus all widgets are directly or indirectly connected to a horizontal and a vertical layout border tabstop. With arbitrary tabstops it is possible to position a widget so that it is not directly or indirectly connected to a layout border tabstop. For example, a widget can be aligned to a tabstop at a two-thirds position in an otherwise empty layout. However, unconnected widgets are not a problem when calculating the constraints for non-overlapping layout specifications. The constraint-generation algorithm described above works for all layouts that are initially non-overlapping. Although a manually added constraint may affect the position of widgets, the generated non-overlap constraints still prevent all widgets from overlapping at all times.

Ensuring Solvable Layouts

There are two types of conflicts that may occur with general constraints. First, a manually added constraint may conflict with other manually added constraints or with those of layout areas, e.g., violate the minimum size constraint of a widget. Secondly, when performing an edit operation, the resulting layout may conflict with manually added constraints. Only hard constraints can cause conflicts and ALE uses these only for minimum sizes and non-overlap. There are several possible strategies to detect and handle conflicting constraints. Here we outline our solution.

By solving the layout after each edit operation on a manual constraint, it is possible to determine whether the constraint causes a conflict. If a manual constraint edit operation triggers a conflict, the simplest solution is to disable the manual constraint. If one of the new edit operations triggers a conflict with a manual constraint, we search for conflicting manual constraints and disable them. We choose to disable conflicting manually added constraints, so that the user can recover from conflicts more easily (Figure 14). This is less drastic than Xcode, where manual constraints are *removed* automatically when a widget is moved to a problematic position. ALE also shows conflicts in the constraint list to make it easier for the user to understand the problem.

In general, isolating conflicting constraints is a hard problem, known as maximum feasible subset (MaxFS). One simple solution is to remove one or more constraints from the system and test whether the conflict exists in the remaining system. This has to be repeated until the smallest set of constraints that makes the layout solvable again has been removed. Since the system has to be solved for each step, this is expensive. However, conflicts are relatively rare, and the common case where no conflict is introduced can be detected by solving only once. A single conflicting constraint can be found in $O(\log n)$ solving steps, using a binary-search-like algorithm. Furthermore, for our new edit operations and manually added constraint edits, all involved constraints and affected tabstops are known. We know which tabstops have possibly caused the conflict and search for conflicting manual constraints that (directly or indirectly) use these tabstops. This simplifies the problem and makes the search more efficient. In practice, the search usually completes in less than 100ms.

PREVIEW OF RESIZE BEHAVIOR

When constructing a user interface, designers usually construct a reasonably sized instance of the layout. Current GUI builders then permit designers to test the window resize behavior manually. However, this is error-prone, as the designer may forget to test all aspects of the resize behavior. A common problem scenario is that the user has connected a widget to a wrong tabstop. An automatic preview of the resized user interface, for both increased and decreased sizes, can reveal

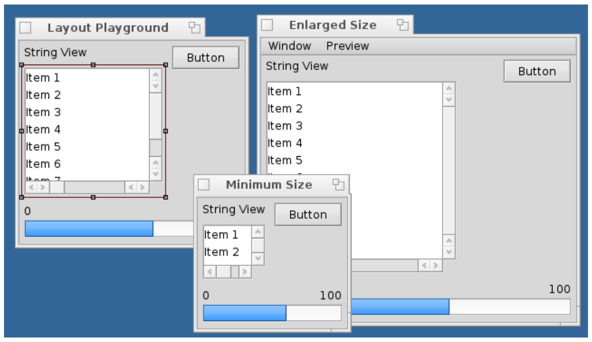


Figure 15. Enlarged and minimum size preview for a layout. The left window shows the preferred size, the bottom the minimum size and the right the enlarged size. Here it becomes visible that the list and the button are not connected to each other.

this. Yet, such preview windows consume precious screen real estate. Also, a small increase or decrease may only yield differences that are not big enough to (easily) see. To address this, we introduce the idea of *optimal resize previews*. The goal of these previews is that the user is able to see the resize behavior of the layout quickly, using minimal screen real estate. Such an optimal resize preview shows the window both at the minimum size and at another size, with a clearly visible size difference relative to the construction window for every widget or constraint, so that the user can see whether the layout specification is correct.

The minimum layout size shows the layout in the most compact form. To determine this size, the hard constraint for the overall layout size is replaced by a strong soft constraint that sets the layout size to zero. This makes the layout as small as permitted by its hard constraints, i.e., minimum size ones. One option for a larger size would be the preferred layout size. However, this is non-optimal, as this size may not significantly differ from the minimum size in all aspects. Thus, we show a second preview that is enlarged enough to enable the user to easily see all size changes relative to the construction window. The just noticeable difference between two simultaneously visible line segments is less than 5 percent of the length [21]. To ensure that our resize visualization is above this threshold, we double the increase by default to 10% and also introduce a minimum increase, at least 10 pixels larger. The user can adjust these two parameters according to preferences and screen pixel densities. To calculate the enlarged size for each widget, we temporarily change the preferred size of every widget to be 10% larger and at least 10 pixels larger than the size in the construction window. Then we compute the size of this modified layout and show it (Figure 15).

DISCUSSION

Here we discuss some of the consequences of the design decisions behind ALE in more detail. ALE uses layout areas to place widgets, and these areas automatically manage all necessary size constraints. This makes editing simpler as all corresponding constraints are hidden from the user. Our use of a quadratic objective function ensures that the preferred size soft constraints are uniquely determined. This makes it impossible to create under-constrained GUIs and makes layouts created with ALE predictable. Also, this approach leads to aesthetically more pleasant layouts [28].

The new edit operations make it easy to create and edit layouts quickly. For example, automatically moving existing widgets aside saves significant time. Also, inserting a widget into the whole available empty space simplifies some tasks. In general, users need to perform fewer steps. Our previous user study confirms this [29]. Since all non-overlap constraints are handled by the system, there is no need to visualize them. There is also no need to display intrinsic size constraints. Thus only manually added constraints are shown, as appropriate. Furthermore, ALE displays tabstops only on demand, and only those that are relevant for the current operation. This leads to a less cluttered design view and permits designers to concentrate on the layout itself.

Moreover, ALE automatically generates non-overlapping layout specifications. This not only reduces the number of constraints that have to be created and maintained, but also reduces the potential for design errors. Tiling the empty space prevents widgets from overlapping, and has the side effect that the extreme layout sizes are well-defined. The designer has some control over this by toggling the tiling algorithm to work horizontally or vertically.

With an increasing number of widgets, the number of tabstops also increases and it may become difficult to select the correct tapstop in a “dense” area. In our experience, this is only rarely the case. One solution is to use snap-and-go [2], as long as each tabstop is at a distinct location. Another is to automatically merge all tabstops at the same location. We will explore whether this leads to unexpected results in future work.

Our current implementation of the active set method is not optimal, as it uses a dense matrix representation. The constraint system has to be solved each time the layout is resized, which happens fast enough. Yet during the edit process the layout has to be solved even more frequently in order to tentatively test a new layout specification. In ALE this usually takes less than 100ms (Intel Core 2 Duo 3GHz), which is sufficient for interactive use and keeps the system responsive and usable at all times. Also, our implementation of the algorithm for generating non-overlapping layout specifications does not guarantee a minimal set of constraints, as some unnecessary constraints may be inserted. Since the runtime performance of an application is more important than the editing process, a pre-solve process could remove redundant constraints.

ALE’s source code is available on github [12]. At a recent open-source meeting (BeGeistert, November 2012) we received much positive feedback on ALE.

CONCLUSION

This paper has presented ALE, a GUI builder that makes it possible to create and edit constraint-based layouts with simple operations. ALE’s new edit operations automatically keep the layout sound and solvable. They also keep widgets aligned relative to each other, which leads to well-structured

[12]`https://github.com/czeidler/haiku`

layouts. Through a new algorithm that automatically generates non-overlap constraints, ALE can create layouts that are guaranteed to be non-overlapping for all possible layout sizes. We have also discussed how ALE's edit operations can be combined with general constraint editing, and how conflicts are resolved. This keeps common layout editing tasks easy, while making arbitrary constraint-based layouts possible. Moreover, we have introduced optimal resize previews to help GUI designers validate the resize behavior of a layout specification. These real-time previews show the layout at a minimum and a sufficiently enlarged size, which optimizes the screen space used for the previews.

In the future, we plan to integrate a sparse matrix representation into the active set solver to accelerate interactions in the GUI builder further. This is a fairly common optimization. Moreover, we plan to add a pre-solver to prune the constraint system further and improve performance. In exceptional cases designers may want widgets to overlap, and we might explore this in the future. Furthermore, the interface for manual constraint editing could be improved, e.g., to make it easier to center widgets globally in the layout, similar to Xcode.

REFERENCES

1. Avrahami, G., Brooks, K. P., and Brown, M. H. A two-view approach to constructing user interfaces. *SIGGRAPH* (1989), 137–146.
2. Baudisch, P., Cutrell, E., Hinckley, K., and Eversole, A. Snap-and-go: helping users align objects without the modality of traditional snapping. CHI (2005), 301–310.
3. Borning, A., Lin, R. K.-H., and Marriott, K. Constraint-based document layout for the web. *Multimedia Syst. 8*, 3 (Oct. 2000), 177–189.
4. Borning, A., Marriott, K., Stuckey, P., and Xiao, Y. Solving linear arithmetic constraints for user interface applications. UIST (1997), 87–96.
5. Fletcher, R. *Practical methods of optimization; (2nd ed.)*. Wiley-Interscience, 1987, ch. 10.3.
6. Galitz, W. *The essential guide to user interface design: an introduction to GUI design principles and techniques*. Wiley, 2007.
7. Gleicher, M. A graphics toolkit based on differential constraints. UIST (1993), 109–120.
8. Hashimoto, O., and Myers, B. A. Graphical styles for building interfaces by demonstration. UIST (1992), 117–124.
9. Heim, S. *The resonant interface: HCI foundations for interaction design*. Pearson, 2007, ch. 6.6.
10. Hudson, S. E., and Mohamed, S. P. Interactive specification of flexible user interface displays. *ACM Trans. Inf. Syst. 8*, 3 (July 1990), 269–288.
11. Hudson, S. E., and Yeatts, A. K. Smoothly integrating rule-based techniques into a direct manipulation interface builder. UIST (1991), 145–153.
12. Hurst, N., Li, W., and Marriott, K. Review of automatic document formatting. DocEng (2009), 99–108.
13. Hurst, N., Marriott, K., and Moulder, P. Cobweb: a constraint-based web browser. Australasian Computer Science Conference (2003), 247–254.
14. International Organization for Standardization. ISO 9241-10: Ergonomic Requirements for Office Work with Visual Display Terminals (VDT) – Part 10: Dialogue Principles, 1996.
15. Jacobs, C., Li, W., Schrier, E., Bargeron, D., and Salesin, D. Adaptive document layout. *Commun. ACM 47*, 8 (Aug. 2004), 60–66.
16. Karsenty, S., Landay, J. A., and Weikart, C. Inferring graphical constraints with Rockit. HCI'92 (1993), 137–153.
17. Lutteroth, C., Strandh, R., and Weber, G. Domain Specific High-Level Constraints for User Interface Layout. *Constraints 13*, 3 (2008).
18. Molich, R., and Nielsen, J. Improving a human-computer dialogue. *Commun. ACM 33*, 3 (Mar. 1990), 338–348.
19. Myers, B. A. User-Interface Tools: Introduction and Survey. *IEEE Software 6* (1989), 15–23.
20. Myers, B. A., and Buxton, W. Creating highly-interactive and graphical user interfaces by demonstration. SIGGRAPH (1986), 249–258.
21. Ono, H. Difference threshold for stimulus length under simultaneous and nonsimultaneous viewing conditions. *Perception & Psychophysics 2* (1967), 201–207.
22. Scoditti, A., and Stuerzlinger, W. A new layout method for graphical user interfaces. IEEE (2009), 642–647.
23. Singh, G., Kok, C. H., and Ngan, T. Y. Druid: a system for demonstrational rapid user interface development. UIST (1990), 167–177.
24. Soltan, V., and Gorpinevich, A. Minimum dissection of a rectilinear polygon with arbitrary holes into rectangles. *Discrete & Computational Geometry 9*, 1 (1993), 57–79.
25. Vlissides, J. M., and Tang, S. A unidraw-based user interface builder. UIST (1991), 201–210.
26. Weber, G. A reduction of grid-bag layout to Auckland layout. Australasian Software Engineering Conference (ASWEC) (April 2010), 67 –74.
27. Zanden, B. V., and Myers, B. A. The Lapidary graphical interface design tool. CHI (1991), 465–466.
28. Zeidler, C., Lutteroth, C., and Weber, G. Constraint solving for beautiful user interfaces: how solving strategies support layout aesthetics. CHINZ (2012), 72–79.
29. Zeidler, C., Stuerzlinger, W., Lutteroth, C., and Weber, G. Evaluating direct manipulation operations for constraint-based layout. INTERACT'13 (to appear).