Evaluating Direct Manipulation Operations for Constraint-Based Layout

Clemens Zeidler[1], Christof Lutteroth[1], Wolfgang Stuerzlinger[2], and Gerald Weber[1]

[1] University of Auckland, 38 Princes Street, Auckland 1010, New Zealand
czei002@aucklanduni.ac.nz, {lutteroth,g.weber}@cs.auckland.ac.nz
[2] York University, 4700 Keele St., Toronto, Canada M3J 1P3
wolfgang@cse.yorku.ca

Abstract. Layout managers are used to control the placement of widgets in graphical user interfaces (GUIs). Constraint-based layout managers are more powerful than other ones. However, they are also more complex and their layouts are prone to problems that usually require direct editing of constraints. Today, designers commonly use GUI builders to specify GUIs. The complexities of traditional approaches to constraint-based layouts pose challenges for GUI builders.

We evaluate a novel GUI builder, the Auckland Layout Editor (ALE), which addresses these challenges by enabling GUI designers to specify constraint-based layouts via direct manipulation using simple, mouse-based operations. These operations hide the complexity of the constraint-based layout model, while giving designers access to its benefits.

In a user evaluation we compared ALE with two other mainstream layout builders, a grid-based and a constraint-based one. The time taken to create realistic sample layouts with our builder was significantly shorter, and most participants preferred ALE's approach. The evaluation demonstrates that good usability for authoring constraint-based layouts is possible.

Keywords: GUI builder, layout editing, layout manager, constraint-based layout, layout preview, evaluation.

1 Introduction

Graphical user interfaces (GUI) are widespread, even on web and mobile platforms. WYSIWYG GUI builders facilitate the creation of such GUIs by developers and designers. In GUI builders, the user can drag & drop widgets from a palette onto a GUI canvas and adjust their properties.

Today's applications are used across a wide range of screen sizes, and users often use multiple applications concurrently on the same screen. Thus users expect that GUIs can be resized and that widgets use the space allocated to an application judiciously. Early drag & drop GUI builders enabled only the creation of static layouts that are unable to adapt to window resizing or other changes to the distribution of screen space. In order to create resizable GUIs with dynamic layouts, a *layout manager* has to be used, which implements a *layout model.* The latter defines how objects in a layout, the *widgets*, can be arranged and how their resize behavior can be specified.

P. Kotzé et al. (Eds.): INTERACT 2013, Part II, LNCS 8118, pp. 513–529, 2013.

Constraint-based layout models are naturally powerful: with the notable exception of flow layouts, many other layout models including gridbag layout [15] can be reduced to constraint-based layouts, see also Figure 1. Many other layout models 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 layouts can align widgets that are situated in completely different parts of a layout [10], which greatly reduces the need for nested layouts.

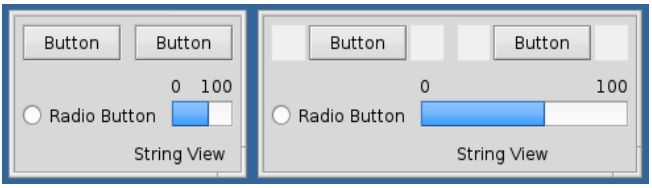

Fig. 1. Example of a constraint-based layout. When resizing, the middle row moves independently of the top and bottom rows. This layout is impossible to achieve with a single gridbag layout, e.g. as the radio button is not constrained to a column.

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. Moreover, specifications may become over-constrained, i.e. have no solution, or permit widgets to overlap each other. Presumably it is for such reasons that support for constraint-based layouts in GUI builders is less developed than for other layout models. This makes it harder for GUI designers to leverage the advantages of constraint-based layouts.

In this paper we evaluate the use of simple drag & drop operations for the manipulation of constraint-based layouts. These operations are sufficient to create a wide variety of constraint-based layouts, including layouts where widgets are aligned to other widgets and no additional constraints exist between them. Our operations are designed to prevent situations where widgets can overlap. To test these operations with regard to their usability, we created a novel GUI builder – the Auckland Layout Editor (ALE) – that implements these operations. In ALE there is no need for the designer to know about the underlying constraint-based layout model.

In two controlled experiments we compared ALE with two state-of-the-art GUI builders. The experiments target the question if ALEs operations are effective for editing of constraint-based layouts. The first experiment compares ALE against a GUI builder that uses the gridbag layout model, which is currently one of the most frequently used layout models. The second experiment compares ALE against a GUI builder that supports constraint-based layouts. The evaluations show the promise of our new approach, with significantly shorter completion times for typical layout creation and editing tasks, and a strong preference from most of the participants for ALE.

Section 2 reviews related work. Section 3 presents ALE and Section 4 the edit operations used in ALE. The evaluation that compares ALE to a gridbag-based and a constraint-based GUI builder is described in Section 5. Section 6 finishes with conclusions.

2 Related Work

There are various non-constraint-based graphical GUI builders. Myers [11] summarized various techniques to create GUIs, including graphical tools to place interface objects on screens. IBuild [14] enabled the creation of complex GUIs in a WYSIWYG manner. It already supported nested layouts and spring-like "glue" layout elements. Moreover, IBuild enabled interactive testing of the resize behavior. Druid [13] predicted the intended alignment and spacing of a widget during editing, facilitating widget placement. FormsVBT [1] supported simultaneous editing of a textual and a graphical representation of a layout.

Some graphical GUI builders permit manual constraint editing. Lapidary [16] supported editing of constraint-based layouts with rich editing functions for constraints. In the constraint-based Gilt system [6], widgets could be aligned relative to user-specified tabstops or other widgets. In contrast to ALE, users had to manage tabstops themselves. In the Intui GUI builder [12] layout constraints can be edited directly by toggling between struts (fixed-size constraints) and springs (variable-size constraints) defined between and inside of widgets. However, only one constraint per widget side is supported. Finally, Rockit [9] automatically infers constraints from static drawings, using hard-coded rules and heuristics. These approaches require the user to edit or handle individual constraints. In contrast, ALE makes it possible to create constraint-based layouts without knowledge of the (somewhat complex) underlying constraint system.

Today, there are many open-source GUI builders, such as WindowBuilder Pro[1], Matisse/Swing[2], and Qt Designer[3]. There are also many commercial GUI builders, such as MS Expression Blend[4], Visual Studio[5], and the Apple Xcode Interface Builder[6]. Most of them support aligning widgets via snapping. With the exception of the Xcode Interface Builder, where Apple added support for constraints in 2012, none of them support the interactive construction or maintenance of constraint-based layouts. In Xcode widgets can be placed "freely" onto the editing canvas, and constraints can be added between the widgets. However, common GUI design guidelines such as [5], which guide designers towards appropriate layouts, generally 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 [7]. ALE automatically keeps widgets aligned, which can lead to a faster layout creation and editing process.

Supple is an automated system that can adapt layouts to changes in display size, in particular to different devices. The system supports discrete changes of widgets, i.e. it changes the controls that are used within an input form depending on the available space [4]. However, the designer has less influence on the actual layout.

A previous usability comparison between constraint-based and grid-based layout models [19] used printed screenshots and sketching. This comparison found that the constraint-based layout model is a competitive alternative to the grid-bag layout model

[1] eclipse.org/windowbuilder
[2] netbeans.org/features/java/swing.html
[3] qt.nokia.com
[4] microsoft.com/expression/products/Blend_Overview.aspx
[5] microsoft.com/visualstudio/en-us
[6] developer.apple.com/technologies/tools

and performs better for editing existing layouts. The current paper compares constraint-based and grid-bag layout model editing directly in interactive GUI builders.

3 The Auckland Layout Editor (ALE)

Similar to other GUI builders, ALE's user interface consists of a component palette and an editing canvas (Figure 2). New widgets can be dragged from the palette into the editing canvas, where the designer can change the layout using a rich set of edit operations, as described in the next section. Here we first define the terms used to describe ALE.

In the underlying constraint-based layout model, each widget has a *minimum*, a *preferred* and a *maximum* size, also called *intrinsic* sizes. A widget should assume its preferred size if there are no other constraints for it, similar to the behavior of a sponge. By default, widgets are surrounded by a small margin, as visible in Figure 1.

There are two common types of linear constraints: *hard* constraints, which have to be satisfied, and *soft* constraints, which may be violated if necessary. When inserting a widget into a layout, some constraints are automatically derived from the intrinsic sizes of a widget: a hard inequality is created for the minimum size, and soft equalities for the preferred and the maximum size. These constraints are added in both the horizontal and vertical direction. As ALE inserts these constraints automatically, users do not need to manage the constraints for intrinsic sizes manually. The preferred size of each widget can be fine-tuned, which is comparable to changing the "weight" of a row or column in a grid-bag layout.

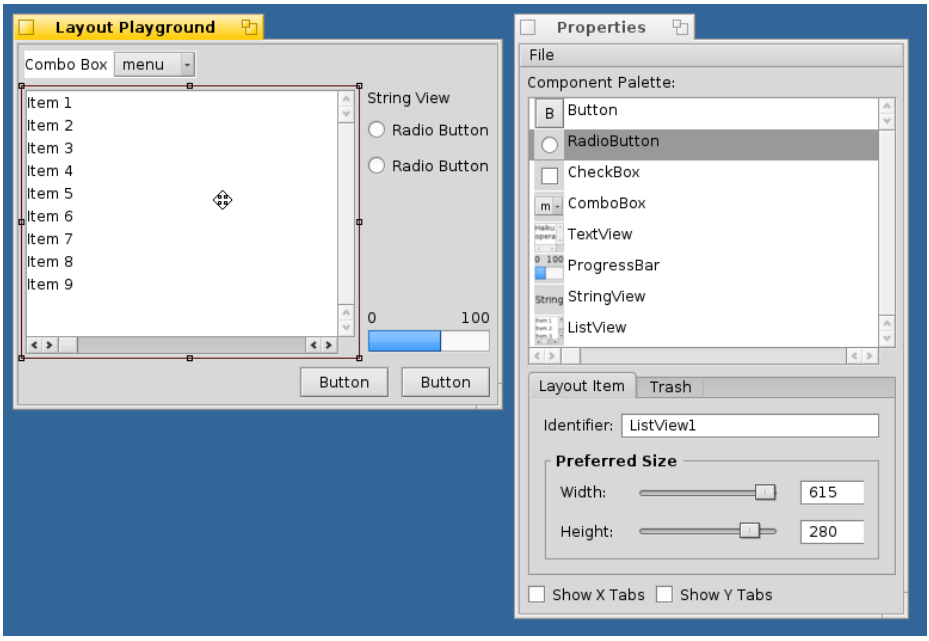

Fig. 2. Screenshot of the ALE GUI builder. New widgets can be inserted into the editing canvas (left) via drag & drop from the component palette (right).

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, snap lines or anchor lines. The edges of each widget are always connected to two horizontal (top and bottom) and two vertical (left and right) tabstops, which form a rectangular *area*. Connecting widgets to the same tabstop aligns them. Figure 3 shows a simple layout and its tabstops. Each GUI container, such as a panel or a window, defines tabstops for its four borders.

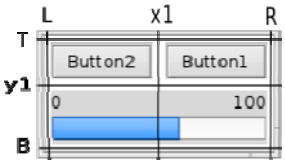

Fig. 3. Widgets are aligned to tabstops. In the picture there are four outer tabstops (L, T, R and B) and two inner tabstops (x1 and y1).

If all widgets are connected to at least one horizontal and one vertical tabstop, the whole layout can be computed directly. The reason for this is that the intrinsic sizes determine the size and the tabstops the x and y position of each widget. The chosen constraint solver controls how soft constraints are handled, which determines the final visual appearance of the layout. In our GUI builder and in the final applications, the quadratic active set method is used for solving [3]. In contrast to a linear approach, this leads to unique and aesthetically more pleasant layouts [17]. An overview of different solving techniques can be found in other works [2, 8].

The minimum size of widgets is guaranteed by a sufficiently large minimum size of the outermost container, e.g. the window. With this, a layout always stays solvable as only a soft inequality constraint is used for the maximum size. If a maximum size constraint is violated, then more space is allocated than a given widget is able to use. In this case, the widget is by default aligned in the center of its allocated space. The right side of Figure 1 illustrates this. The two buttons at the top share the full window width, and the maximum width constraints of the buttons are violated here. Thus, the buttons are centered in their allocated space, as visualized by the light gray areas.

4 ALE Layout Edit Operations

Ideally, a GUI builder should make the creation of a layout easy to learn, simple to achieve, and quick to perform. ALE's edit operations for constraint-based layouts automatically keep widgets aligned to each other as far as possible, by connecting them to existing tabstops. This makes it easier to rearrange widgets while keeping the layout consistent. Furthermore, the edit operations leave a widget connected to at least one horizontal and one vertical tabstop. This ensures that the position of a widget is always well-defined. The edit operations provided are moving, swapping, resizing, inserting, and removing of widgets. These operations enable the user to create complex layouts without manual constraint authoring.

As in most GUI builders, a GUI can be created and edited by drag & drop operations. Edit operations are started by dragging an already placed widget or the border of a widget. Then, ALE checks at each mouse position if an operation can be applied and gives corresponding visual feedback (Figure 4). For this, the operation is applied tentatively in the background. If an operation is appropriate, the user can commit the operation by "dropping" the dragged item 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 such changes, a short animation visualizes how the affected widgets are changed.

Moving a Widget. While dragging a widget, its shape is visualized as a dotted rectangle. We limit the size of the dragged outline to a reasonable size to avoid problems with very large widgets. Two cases need to be considered when moving a widget from one position to another.

First, a widget can be inserted into an empty area (Figure 4). To identify the correct empty area, we attempt to snap the dragged rectangle to existing tabstops. If the widget can only be snapped on one side in a certain direction, horizontally or vertically, a new tabstop is inserted at the opposite side. If a widget cannot be snapped to any tabstop in a certain direction, then two other suitable tabstops have to be found. These are the respective tabstops of the largest empty rectangular area at the pointer position.

The combination of snapping widgets to existing tabstops and placing them into the largest empty area at the pointer makes the move operation quite versatile. For example, it is possible to place a small item accurately into the corner of an empty area by snapping it on two borders. Dropping a widget roughly in the middle of an empty area will fill the area with the widget. Furthermore, a widget can, e.g., be placed in the left half of an empty area by moving the item's left border close to the area's left border midway between top and bottom. ALE visualizes the location where a widget will be placed when "dropped" with a green rectangle (Figure 4).

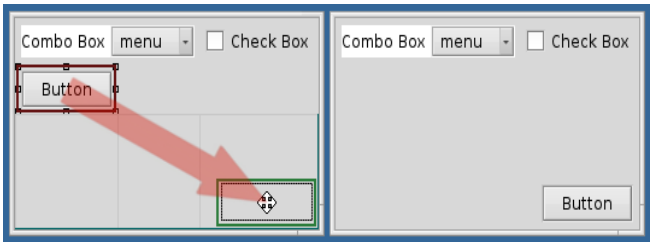

Fig. 4. Move operation: Dragging the button to the bottom-right of the empty area. The area where the button will be inserted when dropped is highlighted in green.

Second, a widget can be placed between an existing tabstop and a widget adjacent to that tabstop. This happens when a widget is dropped close to an existing tabstop and on the adjacent existing widget (Figure 5). In the following, only the insertion at a vertical tabstop is described; the horizontal case works analogously. When dropping the dragged widget, it is inserted so that its top and/or bottom are connected to those of the existing widget, using the same rules as in the first case: if the top or bottom of the dragged rectangle is close to the top or bottom of the existing widget, then they

are snapped together. If only one side is snapped, either top or bottom, a new tabstop is created at the opposite side. If the dragged rectangle is not close enough to the top or bottom of the existing widget, then the dragged widget is connected to both the top and bottom of the existing widget. In Figure 5, the string view is snapped to the bottom of the right button. The left and right tabstops of the moved widget are then set so that the widget is placed between the vertical tabstop and the existing widget.

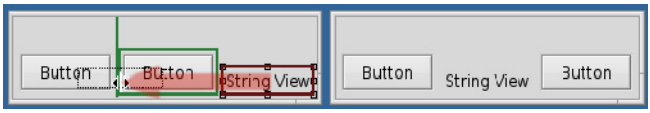

Fig. 5. Move operation: Moving a string view between two buttons

Swapping Two Widgets. This operation is triggered by dropping a widget onto another existing widget. It simply swaps the position of the two involved widgets. Here it is sufficient to connect the moved widget to the tabstops of the other widget, and vice versa.

Resizing a Widget. Dragging one of the borders or corners resizes a widget and allows the user to connect the dragged borders to different tabstops. During resizing, all relevant tabstops are visualized as light blue lines to aid alignment. A resize operation is aborted if an enlarged widget would overlap other widgets.

There are two cases to consider for resizing. First, a widget can be resized to an existing tabstop, by dragging and snapping it to said tabstop (Figure 6). Second, an item can be resized to match its preferred size when a dragged border is released without snapping it to a tabstop. In this case, a new tabstop is inserted for the dragged border and its position is calculated so that the widget gets its preferred size. Finally, the resize behavior of widgets can be controlled manually via the preferred size settings in a properties window (see right side of Figure 2).

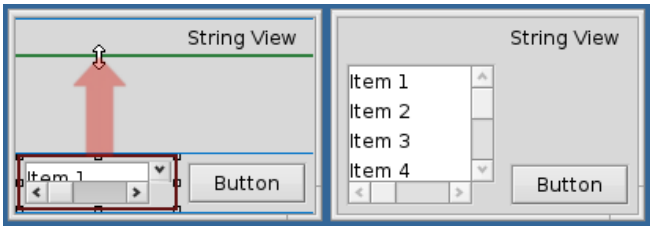

Fig. 6. Resizing the top of the list widget to the bottom tabstop of the string view

Inserting and Removing Widgets. Inserting a new widget is practically identical to the move operation. A widget can be removed from the layout by dragging it outside of the layout and dropping it.

4.1 Filling Gaps

A move, resize or remove operation can disconnect adjacent widgets that were only connected through the one widget that was edited. In this case, a gap may appear in the layout (Figure 7) and the position of some widgets may become undefined in the constraint system; the widgets are "floating".

ALE avoids such situations by checking for disconnected widgets after move, resize and remove operations. If the layout contains widgets or groups of widgets that are disconnected, then these widgets are moved one after another into the direction of the removed or resized widget. If a group was connected to the right side of the removed or resized item, the group is moved to the left, and similarly for other directions (Figure 7). All disconnected groups are moved until they are connected directly or indirectly to at least one horizontal and one vertical layout border.

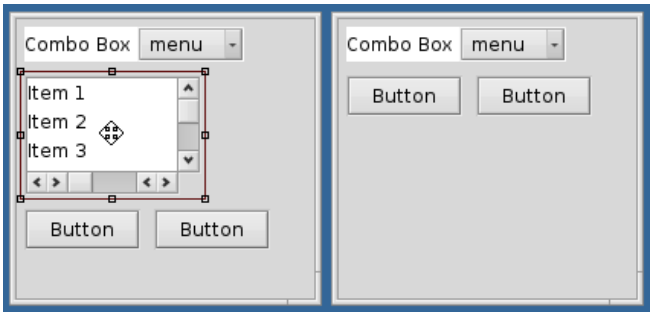

Fig. 7. Filling gaps: Upon removal of the list widget (left), the two buttons are both "floating" vertically. Moving the group of the two buttons to the top fills the gap. The top of the buttons is now connected to the bottom tabstop of the combo box (right).

4.2 Non-overlapping Layouts

ALE guarantees that all created layouts are non-overlapping. In other words, widgets do not intersect each other, regardless of the window size. This ensures that all widgets are completely visible and accessible at all GUI sizes. Non-overlap for a layout also implies that no widget intersects the boundaries of the GUI. Note that we need to 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. For example, the layout in Figure 8 is non-overlapping, but as the layout size is reduced, the check boxes overlap the buttons due to a missing constraint.

ALE's operations are designed to avoid overlap. For this, the layout is tested after each operation for possible overlap. If necessary, additional constraints are added. For example, if the user resizes the layout in Figure 8 during the build process as shown, ALE would add a constraint that keeps a positive distance between the checkbox and the buttons.

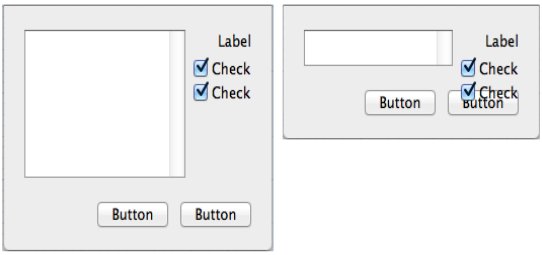

Fig. 8. The constraint-based layout on the left appears sound. Yet, when decreasing its size, as shown on the right, overlap between widgets occurs due to a missing constraint. The used constraint-based GUI builder, Apple Xcode, gave no indication of this issue.

5 Evaluation

We considered two hypotheses, addressed in the following Experiments 1 and 2:

H1 ALE makes it easier to create layouts from scratch compared to other available GUI builders.
H2 ALE makes it easier to edit existing layouts compared to other available GUI builders.

Experiment 1 compared layout creation in ALE with a popular gridbag GUI builder (**H1**). Experiment 2 compared ALE with a modern constraint-based GUI builder, considering both creation of new layouts and editing of existing layouts (**H1** and **H2**).

5.1 Experiment 1: Comparison with a Gridbag GUI Builder

In the first experiment, we targeted **H1** by investigating how our approach performs in comparison to a state-of-the art GUI builder. For this comparison we chose the GUI builder in MS Visual Studio 2010 (VS) as a representative for the state of the art, since it was popular at the time of the study. A prior study compared the usability of the gridbag and the constraint-based layout model, finding that the gridbag layout model has advantages in the creation of new layouts [19]. This supports our choice of a gridbag GUI builder such as VS for comparison with ALE when investigating **H1**.

16 participants, mostly software engineering students with experience in GUI development, were asked to perform four GUI creation tasks, each either with ALE or with VS. In each task, they were asked to rebuild a realistic GUI layout from a sample screenshot. Figures 9, 10 and 11 show the four tasks. We measured task completion time as an indicator of efficiency, and used a post-questionnaire to determine participants' preferences.

With the VS GUI builder it is not easy to modify the row- and column-span in a gridbag layout, since this cannot be done visually. It can only be achieved by opening a properties dialog. Thus, participants were instructed to nest multiple gridbag layouts, permitting users to recreate the target layouts more easily.

For both ALE and VS, a training task was given before the respective main tasks to ensure a reasonable amount of training with both tools. To counteract potential learning effects, half the participants were allocated to a group which first performed the training and tasks I and II with ALE, and then the training and tasks III and IV with VS. The second half used the tools in the opposite order.

If there were errors in the layout after the participants had finished a task, e.g. a widget was placed erroneously, the experimenter indicated the errors to them. Afterwards, timing continued and the participants had to fix these errors. In this way, all participants were able to succeed in all tasks.

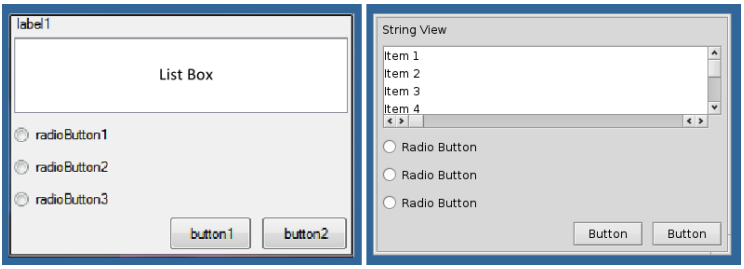

Fig. 9. Experiment 1, Task I: VS on the left and ALE on the right side

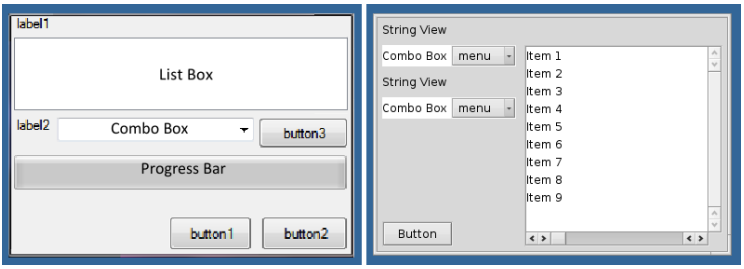

Fig. 10. Experiment 1: Task II for VS on the left and Task III for ALE on the right side

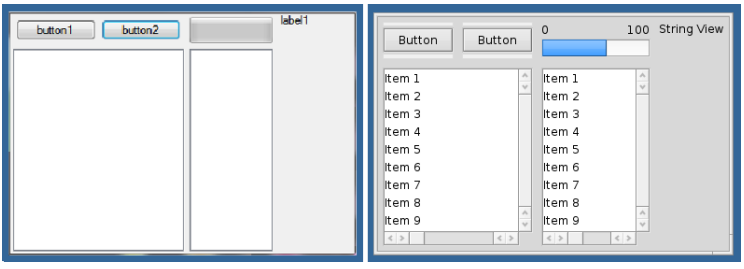

Fig. 11. Experiment 1, Task IV: VS on the left and ALE on the right side

Results and Discussion. The measurements were not normally distributed. The medians of ALE and VS were 74 and 188 seconds, respectively. A Wilcoxon signed-rank test identified a significant effect of the GUI builder (Z = -5.31, $p < 0.0001$), which supports **H1**. Pairwise Wilcoxon signed-rank tests show that ALE

was significantly faster than VS for every task, with $p < 0.01$ or better. Figure 12 shows the individual times broken down by task. According to the post-questionnaire, 11 of the 16 participants preferred ALE over VS. A separate study is necessary to determine what exactly made ALE perform better than VS. It was not the aim of our current work, as we only aimed to evaluate how ALE competes with the gridbag-based VS builder at the task level.

One potential threat to validity is the fact that in VS participants did not use a single gridbag layout, but a nested gridbag layout with a column- and row-span of one. According to observations during the experiment, many participants had difficulties when nesting multiple layouts to create the desired outcome with VS, even though this was easier in the VS GUI builder compared to creating a single gridbag layout. A possible explanation is that a gridbag layout specification has to be understood more thoroughly upfront, and cannot easily be developed on the fly during the design process as with a constraint-based layout approach.

Note that for this experiment, a slightly older ALE version was used [18] than for Experiment 2. Although this version was less polished and had some small usability problems, all operations discussed here were available. It is possible that the different versions of ALE would differ slightly in their performance.

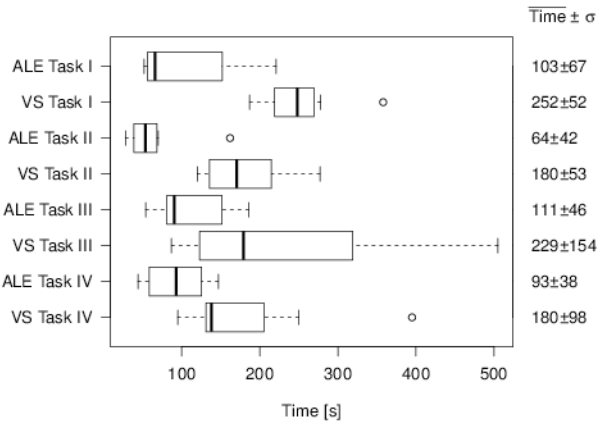

Fig. 12. Boxplot of the task completion times for all four tasks of Experiment 1

5.2 Experiment 2: Comparison with a Constraint-Based GUI Builder

In a second experiment, we tested both **H1** and **H2**. Apple's Xcode offers currently the only easily available GUI builder for constraint-based layouts. Xcode's layout model is similar to ALE's layout model, supporting simple linear hard and soft constraints. Also, Xcode permits free placement and resizing of widgets on the editing canvas. Consequently, we compared ALE with Xcode in this experiment.

The evaluation involved two main tasks (Task V and VI), preceded by a training task that was similar to the main tasks. Each participant performed all tasks once with ALE and once with Xcode. To eliminate order effects, half of the participants started

with ALE, the other half with Xcode. Each task was divided into a layout creation subtask and three editing subtasks.

The first editing subtask (a) required swapping two widgets; the second editing subtask (b) required moving a widget to a position between two other widgets (see Figure 5). For these two subtasks, we expected ALE to perform better because users need only a single operation, while in Xcode multiple operations are necessary. With the free placement approach of Xcode, moving a widget between two other widgets requires the user to first move at least one of the other widgets aside to make room for said widget. Furthermore, it is necessary to manually fill the empty space that the moved widget has left behind. The last editing subtask (c) was more complex and required a combination of multiple edit operations. Figure 13 shows the layout for the creation subtask for Task V, and Figure 14 shows the editing subtask (c) for Task VI. In Xcode, participants were asked to align widgets as well as possible via the snapping functionality provided by the builder. As with the previous experiment, after each task, timing stopped, the experimenter pointed out layout errors to the participants, timing continued and participants had to fix the errors.

After finishing the tasks using Xcode and ALE, the participants were given a Likert-scale questionnaire to gather general information and to analyze their preferences. Furthermore, they were asked for comments in an open-ended question.

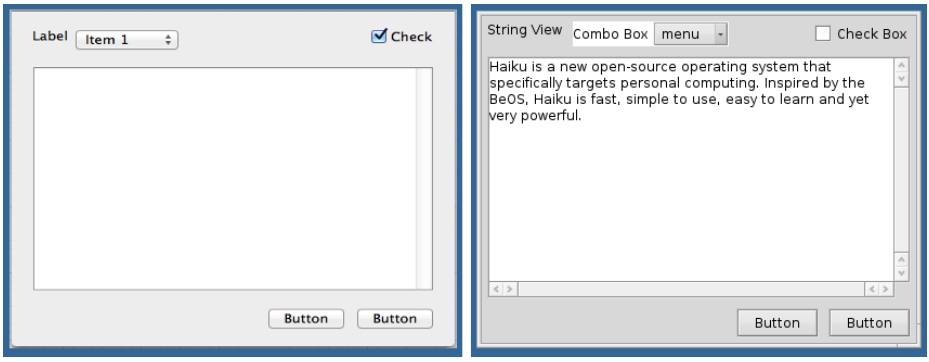

Fig. 13. Experiment 2, Task V: Xcode on the left and ALE on the right side

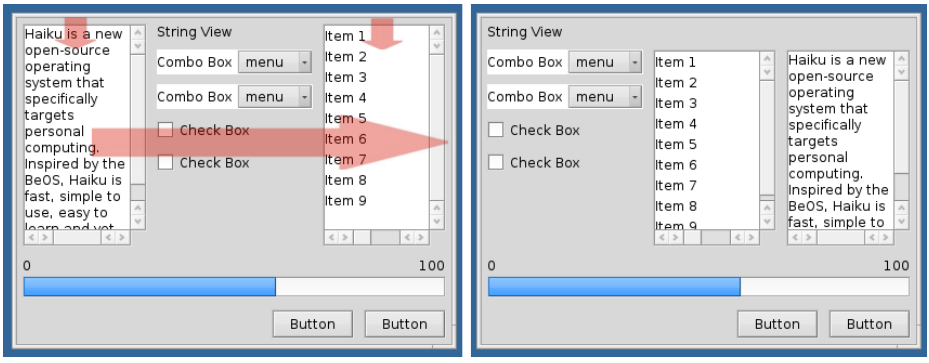

Fig. 14. Experiment 2, Task VI (c): The complex editing subtask

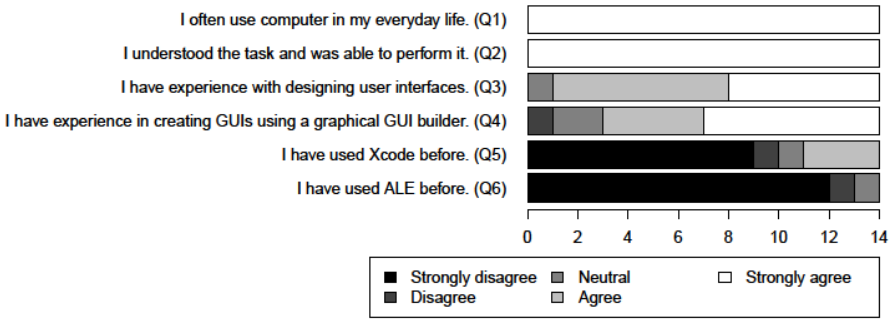

Fig. 15. Results for the general Likert-scale questions of Experiment 2

Results and Discussion. For this study, 14 participants were recruited, mostly graduate Computer Science students. All of them stated that they had understood the tasks. Most participants had experience with user interface design and were familiar with creating GUIs using a GUI builder. Four participants had used Xcode before, while only one participant had used ALE before (see Figure 15).

Overall, the measured task completion times were not normally distributed. The results for the creation tasks are shown in Figure 16.

The medians of all creation times for ALE and Xcode were 66 and 80 seconds, and for all editing times 9 and 38 seconds, respectively. Figure 17 depicts the results for the editing subtasks of Task V. The results for the editing subtasks of Task VI are shown in Figure 18. A Wilcoxon signed-rank test identifies a significant effect of the GUI builder for creation (Z = -2.05, $p < 0.05$), which supports **H1**. There is also a significant effect for editing tasks (Z = -9.19, $p < 0.0001$), which supports **H2**. For layout creation and layout editing, ALE was clearly faster for Task V and Task VI. Pairwise Wilcoxon signed-rank tests show that ALE was significantly faster than Xcode for every creation and editing subtask, with $p < 0.01$ or better. The swapping (a) and the moving (b) subtasks were much faster using ALE.

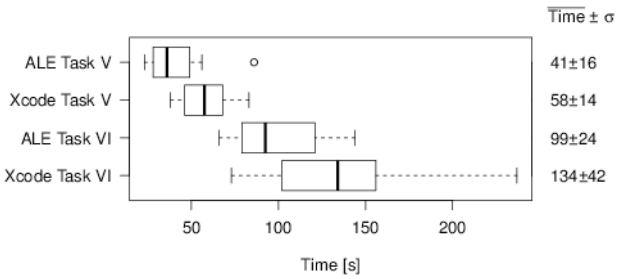

Fig. 16. Task completion times for layout creation in Experiment 2, Task V and VI

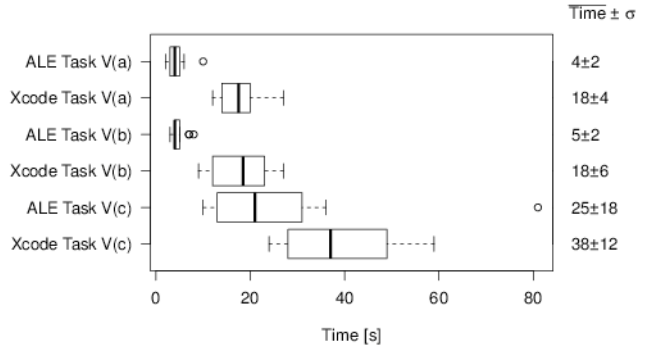

Fig. 17. Task completion times for layout editing in Experiment 2, Task V

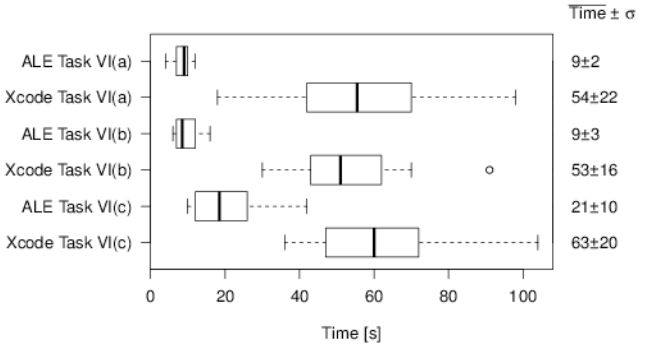

Fig. 18. Task completion times for layout editing in Experiment 2, Task VI

The results from the post-experiment questionnaire show a consistently positive response (Figure 19). Most participants preferred ALE and found it easier for creating and editing layouts. Furthermore, participants enjoyed ALE more than the Xcode builder.

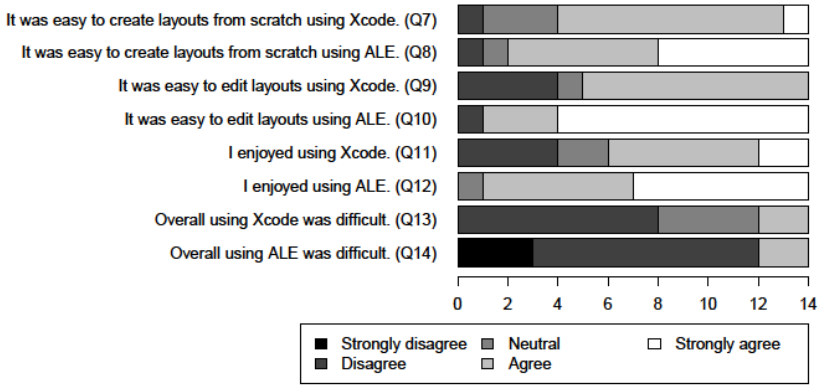

Fig. 19. Results for the Likert-scale usability questions of Experiment 2

Several participants commented in the open-ended question that they liked the swap operation and that layout editing was easier. Other comments pointed out that "*one first had to get used to the different concepts of ALE, e.g. that [a widget] cannot be placed freely.*" Another stated: "*I would imagine that ALE may perhaps be very efficient when acquainted with.*" This is consistent with our observations that participants needed more time to get used to ALE. However, after the training phase most participants were able to perform all tasks without problems.

One noteworthy observation is that with the free placement approach of Xcode it was more difficult to align items precisely. Participants made many erroneous alignments or seemed to be unaware of misalignments in the layout. Another observation is that when using Xcode, participants often first aligned a newly inserted or modified widget precisely to some other widgets, and realized only later that they had to align said widget again to achieve the target layout. ALE avoids this problem by automatically keeping widgets aligned, which is one contributor to the much shorter completion times. A threat to validity is that in Xcode, resizing a text or list widget was sometimes difficult, as users had difficulties clicking the resize handles. However, users also experienced similar minor usability problems in ALE.

Another threat is that layouts in both experiments were relatively small. Since the evaluation took already about an hour we tried to keep the layouts small. However, we believe that the layouts have a reasonable size because for larger layouts one would naturally start to use nested layouts.

6 Conclusion

We presented ALE, a GUI builder that makes it possible to quickly create and edit constraint-based layouts with simple mouse operations. ALE defines a set of layout edit operations that hide much of the complexity of constraint-based layouts from the designer. These edit operations are moving, swapping, resizing, inserting and removing of widgets. All operations maintain alignment and establish appropriate constraints automatically. ALE automatically aligns widgets to each other when placing a widget. It also moves other widgets aside if necessary. Swapping of widgets does not require manual resizing or moving of the adjacent widgets. Additionally, ALE automatically keeps layouts non-overlapping.

In two comparative evaluations, we found that ALE permitted participants to construct several realistic layouts significantly faster than with current commercial solutions, both for a gridbag and a constraint-based layout. Furthermore, we found that editing existing constraint-based layouts is also significantly faster with ALE. Participants enjoyed using ALE, and once familiar with the new edit operations, found it easier to use.

This evaluation demonstrates that it is feasible to utilize the power of constraint-based layouts in graphical GUI builders. The encouraging results from the experiments illustrate also that operations that automatically keep widgets aligned can result in a substantial boost in productivity. Overall, we see this as an indication that there is ample potential for improvements in today's GUI builders.

Future Work. Ideally, a GUI builder should be able to guarantee that all created layouts are non-overlapping, regardless of window size. ALE addresses this by automatically adding constraints when potential overlap is detected. We will describe this functionality in more detail in future work.

ALE's edit operations are already powerful enough to create many common layouts. However, a constraint-based system can be used to create far more complex layouts. For that, general constraint editing has to be integrated into ALE. We will investigate this in the future as well.

Acknowledgements. We would like to thank Rishika Mukerjee and Keerthana Puppala for helping to design and perform the first experiment.

References

1. Avrahami, G., Brooks, K.P., Brown, M.H.: A two-view approach to constructing user interfaces. SIGGRAPH Comput. Graph. 23(3), 137–146 (1989)
2. Badros, G.J., Borning, A.: The Cassowary Linear Arithmetic Constraint Solving Algorithm: Interface and Implementation. Tech. rep., ACM Transactions on Computer Human Interaction (1998)
3. Fletcher, R.: Practical methods of optimization, 2nd edn., ch. 10.3. Wiley Interscience (1987)
4. Gajos, K., Weld, D.S.: Supple: automatically generating user interfaces. In: Proc. 9th International Conference on Intelligent User Interfaces, IUI 2004, pp. 93–100. ACM, New York (2004)
5. Galitz, W.: The essential guide to user interface design: an introduction to GUI design principles and techniques. Wiley (2007)
6. Hashimoto, O., Myers, B.A.: Graphical styles for building interfaces by demonstration. In: Proc. 5th Annual ACM Symposium on User Interface Software and Technology, UIST 1992, pp. 117–124. ACM (1992)
7. Heim, S.: The resonant interface: HCI foundations for interaction design, ch. 6.6. Pearson/Addison Wesley (2007)
8. Jamil, N., Müller, J., Lutteroth, C., Weber, G.: Extending linear relaxation for user interface layout. In: IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece (2012)
9. Karsenty, S., Weikart, C., Landay, J.A.: Inferring graphical constraints with Rockit. In: Proc. INTERACT 1993 and CHI 1993 Conference on Human Factors in Computing Systems, CHI 1993. ACM (1993)
10. Lutteroth, C., Strandh, R., Weber, G.: Domain Specific High-Level Constraints for User Interface Layout. Constraints 13(3) (2008)
11. Myers, B.A.: User-Interface Tools: Introduction and Survey. IEEE Software 6, 15–23 (1989)
12. Scoditti, A., Stuerzlinger, W.: A new layout method for graphical user interfaces. In: 2009 IEEE Toronto International Conference on Science and Technology for Humanity (TIC-STH), pp. 642–647. IEEE (2009)
13. Singh, G., Kok, C.H., Ngan, T.Y.: Druid: A system for demonstrational rapid user interface development. In: Proc. 3rd Annual ACM SIGGRAPH Symposium on User Interface Software and Technology, UIST 1990, pp. 167–177. ACM (1990)

14. Vlissides, J.M., Tang, S.: A unidraw-based user interface builder. In: Proc. 4th Annual ACM Symposium on User Interface Software and Technology, UIST 1991, pp. 201–210. ACM (1991)
15. Weber, G.: A reduction of grid-bag layout to auckland layout. In: 2010 21st Australian Software Engineering Conference (ASWEC), pp. 67–74 (April 2010)
16. Zanden, B.V., Myers, B.A.: The Lapidary graphical interface design tool. In: Proc. SIGCHI Conference on Human Factors in Computing Systems: Reaching Through Technology, CHI 1991, pp. 465–466. ACM (1991)
17. Zeidler, C., Lutteroth, C., Weber, G.: Constraint solving for beautiful user interfaces: how solving strategies support layout aesthetics. In: Proc. 13th International Conference of the NZ Chapter of the ACM's Special Interest Group on HCI, CHINZ 2012, pp. 72–79. ACM (2012)
18. Zeidler, C., Lutteroth, C., Weber, G., Stürzlinger, W.: The Auckland layout editor: an improved gui layout specification process. In: Proc. 13th International Conference of the NZ Chapter of the ACM's Special Interest Group on Human-Computer Interaction, CHINZ 2012, pp. 103–103. ACM, New York (2012)
19. Zeidler, C., Müller, J., Lutteroth, C., Weber, G.: Comparing the usability of grid-bag and constraint-based layouts. In: Proc. 24th Australian Computer-Human Interaction Conference, OzCHI 2012, pp. 674–682. ACM, New York (2012)