Modular Specification of GUI Layout Using Constraints

Christof Lutteroth, Gerald Weber
Department of Computer Science
The University of Auckland
38 Princes Street, Auckland 1020, New Zealand
{lutteroth, g.weber}@cs.auckland.ac.nz

Abstract

The Auckland Layout Model (ALM) is a novel technique for specifying layout. It generalizes grid-based layouts as they are widely used for print layout as well as for GUI layout. Qualitatively, in ALM the focus switches from the cells of the grid to the tabstops between cells. Quantitatively, the model permits the specification of constraints based on linear algebra, and an optimal layout is calculated using linear programming. ALM provides several advantages for developers: first, it supports several different levels of abstraction through higher-level layout constructs that are automatically translated into the lower-level primitives of linear programming. The formalism of linear programming defines a clean separation of ALM's interface and its implementation. Second, the compositional nature of ALM allows developers to group parts of a specification that belong naturally together, resulting in a modular GUI specification. Our experience has shown that it is much harder to achieve a similar separation of concerns when using common GUI layout techniques.

1 Introduction

In today's software engineering, there exist many different technologies for the implementation of GUIs. A lot of effort has been spent in order to improve GUIs, but creating a GUI is often still a difficult task. GUIs are commonly created with the help of GUI toolkits, i.e. libraries which define the various controls and functions that are needed in many GUIs. Often developers have to set the location and size of GUI controls manually, and write code that manages these values during the runtime of an application. For example, if the size of a window changes, an application would typically reposition and resize the controls on that window. In order to make GUI layout easier modern GUI toolkits also include layout engines. Instead of having to take care of the location and size of every control, developers can feed a layout engine with more abstract information, and the layout engine will then position and size the controls whenever this is necessary.

A modern toolkit usually contains several different layout engines, some only supporting very simplistic layouts and others more sophisticated ones. For example, a row layout would take a list of controls and arrange them onto a panel row-wise, starting a new row when the current one has insufficient space. More complex layout engines would typically arrange all controls in a table-like manner, and possibly offer additional constraints. For example, a developer may be able to specify upper and lower limits for the size of each control, or set a preferred size. The layout engine would then try to stretch or squeeze the controls so that they fit into their allocated space without violating the constraints. In this article we present the Auckland Layout Model (ALM), which is a layout engine that can be used for sophisticated layouts.

In this paper we demonstrate that ALM enables GUI layout specifications that are modular, defined on an appropriate level of abstraction, and reusable. Common GUI layout techniques usually fall short of these requirements, which is why GUI layout usually cannot be reused but has to be specified over and over again. From a developer's point of view, ALM offers several advantages: the compositional nature of constraints makes it possible to separate different concerns in a GUI into modules, manage them separately and recombine them later. If some information is not relevant for a layout, it needs not be specified, giving the layout engine more flexibility. For example, the order of the elements in a GUI may be left partially undefined. ALM fills in sensible default values where necessary, so that developers do not need to bother about details that are deemed insignificant. Developers can use arbitrary linear constraints to specify a layout, but can also use higher levels of abstraction, such as higher-order constructs for tables with rows and columns.

The ALM based on linear constraints stands for a whole class of tabstop-based layouts. In principle, we are not restricted to using linear constraints. Any layout model that extends the Auckland Layout Model by allowing for additional (e.g. nonlinear, or integer programming) constraint types is called an *extended Auckland Layout Model*. Also, all layout model that use basically the same qualitative layout model, especially the tabstop approach, are called *Auckland-style layout models*.

Section 2 sums up the basic concepts of ALM, and Sect. 3 illustrates these concepts by explaining some concrete examples. Section 4 discusses aspects of GUI implementation, showing that ALM's flat containment hierarchy allows developers to handle many complex layouts better than the deep hierarchies used by many existing approaches. Section 5 discusses ALM's capability of layout decomposition. Section 6 sums up performance results. Section 8 discusses related work. The article concludes with Sect. 9.

2 The Auckland Layout Model

As mentioned, ALM offers several different levels of abstraction. On the lowest level, ALM is based on linear programming [20], i.e. on linear constraints and the minimization of a linear objective function. The problem of linear programming is formally well-defined, therefore it constitutes a kind of stable interface for the implementation of ALM: it can be implemented on any linear programming solver. On top of this very basic but complete interface, ALM offers higher-order constructs with features typically used for GUI layout: soft constraints, i.e. constraints that may be violated if necessary; abstractions for rectangular areas, which contain controls and offer parameters for preferred sizes, alignment and padding; abstractions for rows and columns, with functionality that enables easy reordering and elision. In the following we will summarize ALM's most important features; a more complete account can be found in [14].

2.1 Linear Constraints

In ALM, the controls of a GUI are aligned at virtual gridlines called *tabstops*, or tabs for short. Tabs are either vertical or horizontal, also known as x- and y-tabs. X-tabs are represented as variables holding an x-coordinate, and the latter ones as variables holding a y-coordinate.

If we consider a layout with x-tabstops $x_0, \ldots, x_m$, $m \in \mathbb{N}$, and y-tabstops $y_0, \ldots, y_n$, $n \in \mathbb{N}$, then a linear constraint for that layout can take the form

$$a_0 x_0 + \ldots + a_m x_m + b_0 y_0 + \ldots + b_n y_n \; OP \; c$$

with the coefficients $a_0, \ldots, a_m, b_0, \ldots, b_n$ and the right side c being real numbers, and the operand OP being one of $\{\leq, =, \geq\}$. A layout can contain an arbitrary number of such constraints; usually most of the coefficients in a constraint are zero. It is possible to use different units for different constraints. A value may be defined in pixels, which makes the actual size dependent on the graphics hardware, or in real-world units like cm, which always produces the same size. In the following sections we will examine different types of linear constraints, and describe how these can be useful for GUI layout. We will only consider equalities, but the concepts can be transferred to inequalities.

2.1.1 Absolute Constraints

We use absolute constraints in order to place x- or y-tabstops at particular x- or x-positions of the UI, respectively, or set the width or height between tabstops to a fixed value. If we want, for example, to set x-tabstop x_3 at position 50, we simply use the constraint

$$x_3 = 50.$$

In order to set the width of the area between x_1 and x_2 to 100, we would use the constraint

$$x_2 - x_1 = 100.$$

Such constraints are a very straightforward way to define the absolute properties of a UI, i.e. the properties that do not change when, e.g., resizing the window the UI is displayed in. Note that absolute constraints may be impossible to satisfy under some circumstances. If, for example, the available display area is only 10cm wide, the width between two x-tabstops cannot exceed this value.

2.1.2 Relative Constraints

In contrast to absolute constraints, relative constraints describe the position of tabstops or proportion of areas relative to others. This is useful in order to adapt the layout to changing circumstances, like UI display size or resolution. The layout engine recalculates the layout when such a change occurs.

Relative constraints can be used in order to position tabstops at positions relative to other tabstops. One might, for example, want to position an x-tabstop x_2 exactly between two other x-tabstops x_1 and x_3. Let us assume that $x_1 \leq x_3$, then the constraint can be expressed as follows:

$$x_2 - x_1 = x_3 - x_2 \Leftrightarrow -x_1 + 2x_2 - x_3 = 0.$$

Similarly, we can center an area that is delimited by x-tabstops x_2 and x_3, $x_2 \leq x_3$, horizontally between two other x-tabstops x_1 and x_4, $x_1 \leq x_4$:

$$x_2 - x_1 = x_4 - x_3 \Leftrightarrow -x_1 + x_2 + x_3 - x_4 = 0.$$

We only need to make sure that the area we want to center does not exceed the boundaries of x_1 and x_4 by specifying that $x_1 \leq x_2$ or $x_3 \leq x_4$.

Another usage for relative constraints is the specification of an area's proportions relative to those of another one. If, for example, we want the width between x-tabstops x_1 and x_2 to be twice as much as the the width between x_3 and x_4, we would use the following constraint:

$$x_2 - x_1 = 2(x_4 - x_3) \Leftrightarrow -x_1 + x_2 + 2x_3 - 2x_4 = 0.$$

Since a constraint can contain x-tabstops as well as y-tabstops, it is also possible to specify the aspect ratio of an area. We could, for example, specify the aspect ratio for an area $(x_1, y_1, x_2, y_2, moviepanel)$, which might contain a control for displaying a video. Because we do not want the video to be shown with an arbitrary, distorted aspect ratio, we could, for example, set the ratio of width and height of this area to 16:9. This would be achieved with the following constraint:

$$\frac{x_2 - x_1}{y_2 - y_1} = \frac{16}{9} \Leftrightarrow -x_1 + x_2 + \frac{16}{9}y_1 - \frac{16}{9}y_2 = 0.$$

The aforementioned constraints are hard, i.e. if they are contained in the specification of a layout, then this layout will satisfy them strictly. Sometimes, however, we want to specify constraints that may not be satisfied fully if circumstances do not permit so. Such constraints are called *soft constraints*; they are natural in applications for user interfaces and have been used in the past [1].

Soft constraints are important in order to prevent overconstrained specifications which would be infeasible otherwise. For example, a specification may become overconstrained when several sets of constraints are merged. Therefore, soft constraints are relevant when layouts are specified in a modular manner, i.e. composed using several partial specifications. ALM supports soft constraints as an abstraction layer on top of the linear programming solver. Soft constraints can be handled in exactly the same manner as hard constraints; they are implemented as a subclass of hard constraints. In addition, they can be prioritized using penalty parameters for positive and negative deviations from their exact solution. Several approaches for prioritizing constraints have been devised, such as constraint hierarchies [5] that make sure that important constraints are satisfied first.

On a certain level, it is advisable to use only soft constraints, so that a specification will always have a solution. The different levels of a constraint hierarchy can be related to access privileges: hard constraints should only be accessible to trusted developers, since they may render the specification infeasible. Similarly, soft constraints with high priority should be chosen carefully since they may detrimentally affect all constraints with lower priority.

2.2 Areas

The controls of a GUI are organized in rectangular *areas*, which are bound by a pair of x-tabstops and a pair of y-tabstops each. In general, an area a is defined as follows:

$$a =_{def} (x_1, y_1, x_2, y_2, content)$$

The x-tabstops x_1 and x_2 delimit the area on the x-axis, with x_1 being to the left or on the same position as x_2; the y-tabstops y_1 and y_2 delimit it on the y-axis, with y_1 being above or on the same position as y_2. $content$ can be a control of the GUI, but can also be $empty$. Empty areas are useful, e.g., for defining margins or padding, or for extending a layout specification with ordinal information. Like this, a set of area definitions result in a partial order on the x- and on the y-tabstops.

One can think of an area as a group of adjacent cells in the table created by the tabstops that are merged into a rectangle to house one of the graphical elements of the UI. This approach covers the common colspan and rowspan features of other concepts for tabular layout: colspan means that adjacent table cells lying on a horizontal line can be merged into a single cell; rowspan means that adjacent table cells lying on a vertical line can be merged, i.e. the resulting cell spans over several rows or columns. ALM generalizes colspan and rowspan by permitting areas that extend between any two x-tabs and any two y-tabs. In addition to this, ALM permits overlapping areas – something which cannot be achieved by merging adjacent cells with colspan and rowspan. This is because tabs may run through areas, and any tab can be used to delimit an arbitrary number of areas.

2.2.1 Additional Area Parameters

ALM supports the specification of parameters for areas that allow the layout engine to find optimal dimensions for them. One of those properties is the preferred size. This value expresses the size that an area would need in order to fulfill its function in an optimal manner. Usually it is based on the size of the content that is displayed in a control. In many GUI toolkits, such values are made available by all controls, and are thus taken directly from them by a layout engine. That way the layout can be adapted immediately when preferred sizes change, e.g. when the content shown in a control changes. Like this, screen real-estate is not unnecessarily wasted, and areas can be used in a modular fashion: each area contains properties describing its demands, while the system as a whole finds a global solution that respects the desires of each area as much as possible. Analogously to soft constraints, the demands of areas can be prioritized, so that areas that are considered more important can be given preference over less important areas.

The following list describes some of the area parameters:

Preferred size s_{pref}**:** the preferred width w_{pref} and height h_{pref} of an area. These optional values can often be calculated by a GUI toolkit, and can therefore be directly taken from there.

Minimum size s_{min}**:** the minimum width w_{min} and height h_{min} of an area. Both values are optional.

Maximum size s_{max}**:** the maximum width w_{max} and height h_{max} of an area. Both values are optional.

Shrink penalty p_{shrink}**:** a coefficient for each dimension that indicates the reluctance on the part of the area to take on sizes that are smaller than the preferred size. This value is optional, and sensible default can be chosen dependent on the type of the control in the area. Controls that contain important information, such as buttons with text, should not be shrunk easily and should therefore have a higher shrink penalty.

Expand penalty p_{expand}**:** a coefficient for each dimension that indicates the reluctance on the part of the area to take on sizes that are larger than the preferred size. This value is optional, and again, a sensible default can be obtained by considering the control type. Controls such as textboxes where users enter data can benefit more from additional space than controls with fixed information content, such as buttons, and therefore the former should have a smaller expand penalty.

Having useful default values for the different parameters of the model makes specifications more compact and hides the complexity of unused model features. A GUI developer needs, for example, only consider penalty coefficients if they want to model something that differs significantly from the default behavior. Like this, developers can learn the features of the model step by step, and can use the model even with a limited understanding.

2.3 Rows and Columns

A useful pattern that enhances maintainability of table definitions is the *separate tabstop* pattern, which means that separate tabstops are used to delimit different logical objects, even if the tabstops end up on the same coordinates. ALM offers abstractions for columns and rows, which contain two x- or y-tabstops each. Instead of using tabstops directly, an area can be arranged by placing it into a particular row and column. The rows and columns offer methods that make it possible to adjust their size and order dynamically. For example, each row has an optional `Previous` and `Next` property referring to the row directly above and below. Rows can be removed from and inserted between existing rows. The order in which the columns appear in the table is encapsulated in a separate set of constraints, managed by ALM and independent of the definition of the columns themselves. Hence, the order of the columns can be changed very easily without interfering with the rest of the specification. In fact, all the parts of a specification, such as areas, rows and columns, also allow developers to add their own constraints which are then managed together with the respective part in a modular fashion.

3 Examples

In order to use ALM in C#, only a few lines of source code have to be added to the definition of the parent control of the GUI:

```
1   ALM.ALMEngine le =
2     new ALM.ALMEngine();
3
4   public override
5   LayoutEngine LayoutEngine
6   { get { return le; } }
```

Lines 1-2 add a field containing an ALM layout engine instance to the control. Lines 4-6 override the `LayoutEngine` property of the control so that the ALM layout engine instance is used. The runtime system automatically calls the `Layout` method of the layout engine whenever the GUI changes.

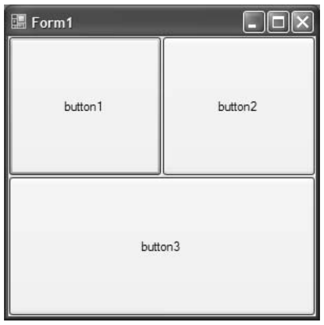


Figure 1. Layout with three areas and two additional constraints.

The layout shown in Fig. 1 is specified in the following manner:

```
LayoutSpec ls = new LayoutSpec();
XTab x1 = ls.AddXTab();
YTab y1 = ls.AddYTab();

ls.AddArea(ls.Left, ls.Top, x1, y1,
  button1);
ls.AddArea(x1, ls.Top, ls.Right, y1,
  button2);
```

```
 9   ls.AddArea(ls.Left, y1, ls.Right,
10     ls.Bottom, button3);
11
12   ls.AddConstraint(
13     new double[] { 2, -1 },
14     new Variable[] { x1, ls.Right },
15     OperatorType.EQ, 0);
16   ls.AddConstraint(
17     new double[] { 2, -1 },
18     new Variable[] { y1, ls.Bottom },
19     OperatorType.EQ, 0);
```

In line 1, a new specification instance is created. In the following, all layout elements are added to this specification. First, two tabs are added, then the three areas containing the button controls `button1`, `button2` and `button3`. The fields `Left`, `Right`, `Top` and `Bottom` of a linear specification contain tabs for the borders of the GUI. The two linear constraints that are added in lines 12-19 define that x-tab `x1` should be halfway between the left and right borders of the GUI, and y-tab `y1` halfway between the top and bottom borders. No matter how we resize the window, the two columns in the GUI will always have the same width, and the two rows will always have the same height.

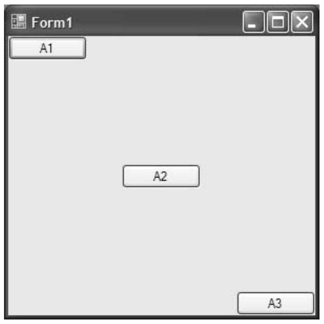


Figure 2. Buttons aligned in three rows.

The next example demonstrates the abstractions for rows and columns, specifying the layout shown in Fig. 2:

```
 1   Column c1 = ls.AddColumn();
 2   Row r1 = ls.AddRow();
 3   Row r3 = ls.AddRow();
 4   Row r2 = ls.AddRow();
 5   r1.Next = r3;
 6   r2.InsertAfter(r1);
 7
 8   Area a1 = ls.AddArea(r1, c1, b1);
 9   a1.HAlignment = HAlignment.LEFT;
10   a1.VAlignment = VAlignment.TOP;
11   Area a2 = ls.AddArea(r2, c1, b2);
12   a2.HAlignment = HAlignment.CENTER;
13   a2.VAlignment = VAlignment.CENTER;
14   Area a3 = ls.AddArea(r3, c1, b3);
15   a3.HAlignment = HAlignment.RIGHT;
16   a3.VAlignment = VAlignment.BOTTOM;
17
18   r2.HasSameHeightAs(r1);
19   r3.HasSameHeightAs(r1);
```

In lines 1-4 a column `c1` and three rows `r1`, `r2` and `r3` are created. The following lines demonstrate how the relation between rows (or columns) can be specified, or dynamically changed if necessary: line 5 causes `r3` to be directly below `r1`, and line 6 inserts `r2` between `r1` and `r2`. Lines 8-16 demonstrate alignment of controls in areas. The last two lines use convenience methods for setting same-height and same-width constraints for rows and columns.

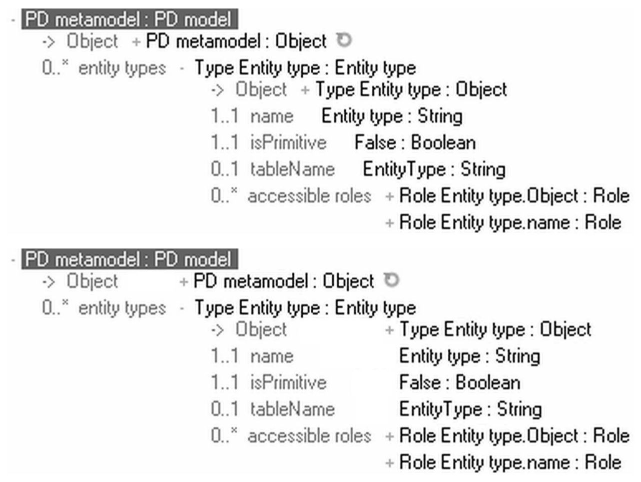
4 Hierarchical vs. Flat Layout Implementation

When programming a GUI, we want the parts of the GUI to be modular, just as we would want with any software component. There should be as little dependencies as possible between individual controls, and each control should implement a clearly defined functionality. If there are dependencies that have to be considered by the programmer, they can drastically reduce the maintainability of the application. This section describes how ALM helps to deal with the controls of a GUI in a modular way.

Modularity on different levels of granularity in a GUI is usually achieved with a recursive, hierarchical grouping of the controls, i.e. the containment hierarchy. This means that most controls have a parent control and are placed in the screen space belonging to that parent control. It also means that many controls can have child controls that are contained in the screen space allotted to them. It is often a parent's responsibility to manage the position and size, i.e. the layout, of their children, so that the parent together with its children can function as a self-contained entity, i.e. a module. Dependencies between children are undesirable, especially if they do not have the same parent. Like this, most nodes of the containment hierarchy can be treated in a modular way.

Some controls naturally have no parent, such as windows. Others cannot have children, e.g. textboxes. Another category of controls is explicitly made to contain other controls, and hardly offers any way of interacting with a user themselves. But such controls, typically panels, often offer functionality for laying out their children according to some scheme. Typical examples are panels that arrange their children in a row, a column, or a table.

Panels are typically used for grouping related controls and maintaining the layout dependencies between them. For this to work the controls between which layout dependencies exist have to be in the same panel. Unfortunately there are cases where the natural containment hierarchy does not group all these controls together. Figure 3 shows such an example.



```
- PD metamodel : PD model
    -> Object     + PD metamodel : Object
    0..* entity types  - Type Entity type : Entity type
                            -> Object  + Type Entity type : Object
                            1..1 name      Entity type : String
                            1..1 isPrimitive    False : Boolean
                            0..1 tableName    EntityType : String
                            0..* accessible roles  + Role Entity type.Object : Role
                                                   + Role Entity type.name : Role

- PD metamodel : PD model
    -> Object            + PD metamodel : Object
    0..* entity types  - Type Entity type : Entity type
                            -> Object          + Type Entity type : Object
                            1..1 name           Entity type : String
                            1..1 isPrimitive    False : Boolean
                            0..1 tableName      EntityType : String
                            0..* accessible roles  + Role Entity type.Object : Role
                                                   + Role Entity type.name : Role
```

Figure 3. Screenshots of a tree view implemented with nested ordinary tables (top) and implemented with ALM (bottom).

The top part of Figure 3 shows a screenshot of a hand-tailored tree view that was implemented and laid out in the typical manner with nested panels. The screenshot is taken from an editor for structured data, and the specialty of the tree view is that there are two types of nodes: the elements set in black text are data elements with names and types, and the elements set in gray text are roles between data types that describe the relations with which the data elements are connected. The screenshot shows the metamodel of the application, i.e. a representation of the types that describe a data model.

As we can see in the screenshot, the data nodes and role nodes have different layouts. Each node has its own panel that contains the GUI representation of the node itself and also all its child nodes. A data node lays out all its role nodes under its own representation, in rows and with an indentation on the right side. A role node lays out its data elements in rows to the right side of its own representation.

The whole tree view is structured recursively, since the panel of a node contains all its subnodes as well. This reflects the structure of the underlying data and makes it possible to treat subtrees of the data in a modular manner. If a data element changes, its node can redraw itself and its subnodes without interfering with the rest of the GUI. But unfortunately, alignment of controls between different nodes, as we see it in the bottom part of Figure 3, is not possible.

The screenshot in the bottom part shows a version of the tree view that was implemented with ALM. The data nodes that are on the same level in the tree are all left-aligned, even though they would be grouped in completely different panels in a nested implementation. The same is the case for the role nodes. If we wanted to achieve this effect with nested panels, we would first have to find the maximal width of all the nodes on each level by going through the tree. Then we would have to set the width of all the nodes on each level to the respective maximum. And each time a node changed, we would have to make sure that the alignment is kept. Implementing this would create very ugly dependencies and make the code of the tree view much harder to maintain.

Layouts specified in ALM may be visually hierarchical, but such hierarchies need not be reflected in the containment hierarchy. Partial specifications can be added incrementally to a single layout specification, which is used to lay out all controls on a single panel. Layout constraints are not expressed in the containment hierarchy because this would preclude other layout constraints, as the example illustrated. Instead, such constraints can be flexibly added or removed from the set of constraints which makes up the specification, and organized freely by the developer. For example, constraints can be grouped so that every control or group of controls encapsulates their constraints in a modular manner.

We call this flat layout implementation, in contrast to the deep containment hierarchy that is typically used to implement a layout. Of course, there still is a containment hierarchy: it is used to construct hierarchical relationships that are natural for some controls, for example, to put controls into a window, or a picture onto a button. However, it is not used to express constraints of the layout, as would be the case with nested panels. Those constraints are cleanly represented in a separate layout specification instead.

5 Model Decomposition

We call the inequality system S for a concrete layout the *layout system* of this layout. Such a layout system has a variable set $V(S)$ of those variables used in its constraints. If we create two different layout systems S, T, then naturally we will keep their variable sets disjoint. Let's assume S describes a complex interface, and we want to use an already prepared layout system T just within a cell of S. Let x_l, x_r, y_l, y_r be the tabstops describing the cell in S, and let $x_{min}, x_{max}, y_{min}, y_{max}$ be the outer boundaries of T. Then the natural way to compose the layout such that T is contained in S is to join the constraints of both layout systems and add the following equations:

$$x_{min} = x_l, x_{max} = x_r, y_{min} = y_l, y_{max} = y_r.$$

There is a different way how this could have been achieved, and this is by modifying T, namely by operationally substituting the variables in T with those from S:

$$x_{min} \rightarrow x_l, x_{max} \rightarrow x_r, y_{min} \rightarrow y_l, y_{max} \rightarrow y_r.$$

Adding equations is preferable over substituting variables from a software engineering perspective. We see that inequality systems can be used as components, and all variables of each component are possible ports of these components to begin with. The set of variables defines an implicit interface of each inequality system. We have a precise notion of glue between such modules: equations between variables of different inequality systems. As we see, our layout provides an elegant example of a component-oriented, domain-specific, non-universal language. The glue language is small, one is tempted to say minimalistic, and less expressive than the component definition language. The composition approach using equations as glue is therefore suitable for a classical file-based software component model. The inequality systems can be defined in separate files and reused without textual modification.

From a software-engineering point of view, visibility modifiers are desirable, so that not all variables of the inequality system are necessarily exposed to the public. There are many operational ways to achieve this. Classical scoping approaches come to mind: variables are annotated as public or private within an inequality system, but this is not the focus of this paper. In the example above we see clearly, how the approach of ALM flattens the interface definition, as it was explained in Section 4.

A natural application of the composition approach to the C# implementation of ALM is the following: inequality systems are made reusable by defining them as a class that defines variables. The constraints are added in the constructor of the class. The layout system for Figure 1 would look as follows:

```
class ExampleLayout{
  public XTab Left, Right, x1;
  public YTab Top, Bottom, y1;

  public ExampleSystem(LayoutSpec ls) {
    x1 = ls.AddXTab();
    y1 = ls.AddYTab();

    ls.AddArea(Left, Top, x1, y1,
       new Button());
    ls.AddArea(x1, Top, Right, y1,
      new Button());
    ls.AddArea(Left, y1, Right,
      Bottom, new Button());
    ls.AddConstraint(
      new double[] { 2, -1 },
      new Variable[] { x1, Right },
      OperatorType.EQ, 0);
    ls.AddConstraint(
      new double[] { 2, -1 },
      new Variable[] { y1, Bottom },
      OperatorType.EQ, 0);
} }
```

Note that in contrast to the earlier version, the fields `Left, Right`, etc. are not members of the LayoutSpec object. Otherwise the definitions remain the same. As we see, we in effect practice metaprogramming here. The layout system specified this way is reusable, it can be reused even several times in the same LayoutSpec. The approach above can be extended to layout system definitions that are parameterised and produce a variable number of variables. In this case, the variables would be returned in an array. The following listing shows an application of this definition. We use the same layout system twice as a subsystem. We assume we define a convenience method `ls.addEq(v1, v2)` that adds an equality $v1 = v2$ to a layout `ls`.

```
LayoutSpec ls = new LayoutSpec();
XTab x1 = ls.AddXTab();
ls.AddConstraint(
  new double[] { 3, -2 },
  new Variable[] { x1, ls.Right },
  OperatorType.EQ, 0);
lsub = new ExampleLayout(ls);
ls.addEq(ls.Left, lsub.Left);
ls.addEq(ls.Top, lsub.Top);
ls.addEq(x1, lsub.Right);
ls.addEq(ls.Bottom, lsub.Bottom);
rsub = new ExampleLayout(ls);
ls.addEq(x1, rsub.Left);
ls.addEq(ls.Top, rsub.Top);
ls.addEq(ls.Right, rsub.Right);
ls.addEq(ls.Bottom, lrsub.Bottom);
```

6 Performance

The performance of the linear programming solver is very important since this computation has to be made, e.g. whenever the end user resizes the top window of an application. The asymptotic complexity of linear programming and the popular algorithms is well discussed in literature. The problem itself is polynomially solvable, as shown by L. Khachiyan and later practically demonstrated by Karmarkar [11]. In practice, variants of the simplex algorithm [8] are popular despite its known exponential worst case behavior.

Although current production-grade linear programming solvers have become fairly efficient, their use in user interface technology can pose new challenges. Operations such as window resizing require the continuous recalculation of the layout with every movement event of the mouse. If we want to perform this recalculation using linear programming, then a very good performance is needed in order to make the movement smooth for the human eye.

In the past this triggered the development of dedicated solvers that use the incremental nature of this problem efficiently, e.g. the Cassowary solver [1]. Nowadays, however, the general purpose linear constraint solvers are very fast if used for incremental problem solution, thus eliminating the need for special purpose solvers. Accordingly, many such solvers are not supported any longer.

Once an initial solution is available, linear programming solvers can efficiently solve a slightly changed linear programming specification in an incremental manner. The initial solution is usually the most expensive step, with the incremental calculations being several times faster. An initial solution can be calculated a priori, thus not impacting the performance of the user interface.

Our implementation of ALM uses the open-source lp_solve linear programming solver [2], employing a simplex algorithm. We have tested the solver on different hardware, and found that even on old hardware layout calculation is fast, resulting in a smooth interactive behavior. The simplistic layouts shown in Figs. 1 and 2 are calculated in about 0.3 milliseconds on a Pentium M with 1.6 Ghz, and in about 0.7 milliseconds on a Pentium 3 with 788 Mhz. The more sophisticated layout in Fig. 4 takes about 0.7 milliseconds on a Pentium M with 1.6Ghz, and about 1.4 milliseconds on a Pentium 3 with 788 Mhz.

The layout illustrated in Fig. 5 contains 100 areas. It was generated randomly, and not all areas are visible due to the lack of screen space. The rigidity parameters are set so that some areas with small rigidity are swallowed up by others. On a Pentium M with 1.6Ghz the layout calculation takes about 6 milliseconds, and on a Pentium 3 with 788 Mhz it takes about 15 milliseconds.

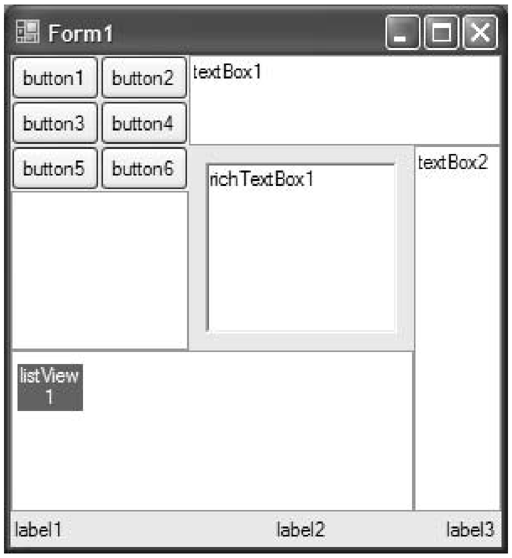


Figure 4. Layout with 14 areas.

7 Grid-Based Approaches to Layout

For the layout of GUIs as well as documents, grid-based approaches have become a widespread solution [12]. In grid-based layouts, the drawing plane is divided into grid cells; we call them simple cells. In the more advanced types

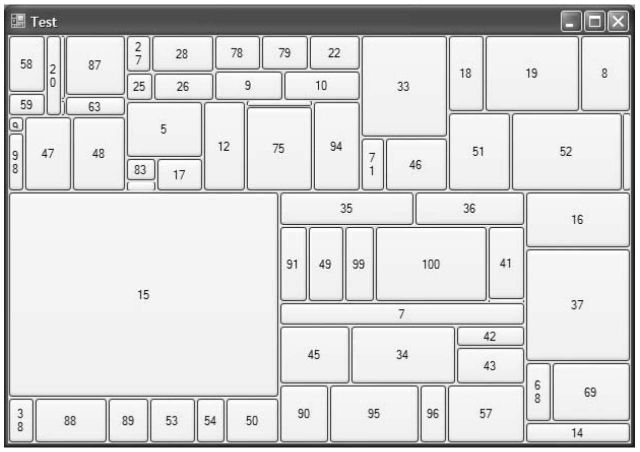


Figure 5. Randomly generated layout with 100 areas.

of this concept, single components can occupy several simple cells. We call all such advanced grid-based layouts *gridbag layouts*, after the respective layout manager in Java, but as an example we will mostly look at HTML tables [18].

In HTML, each cell element has modifiers *colspan* and *rowspan* that creates non-simple cells through the union of adjacent rows or columns. These integer attributes specify the number of joined simple cells in each direction. This approach requires to fix the total order of gridlines (in our terminology: tabstops) in each dimension. This has characteristic disadvantages. We focus here on two problems closely related to each other. The first is a classical software engineering problem and shows that gridbag layouts can lead to bad design through unnecesary coupling. The second problem is that a gridbag layout is less powerful than ALM when it comes to adaptive resizing behavior of the layout.

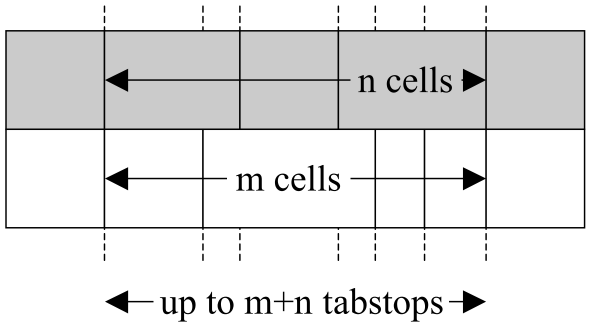


Figure 6. Two rows with different internal layout.

We discuss the problems for the x-dimension; for the y dimension, the same applies. A *colspan* attribute is used if the interval in x-direction used by a cell in one row will be subdivided in some other row. Take the second cell in the first row in Fig. 6. In the row below, there is a boundary between two cells that is not extending to the first row. Hence said cell in the first row will need a *colspan* value of at least 2. The first problem with that is that the *colspan* definition for one cell is influenced by some unrelated activity. Creating a new row with a peculiar layout makes it necessary to touch potentially all other rows. This is illustrated in Fig. 6 with two rows. In the worst case, all cells in the upper row are influenced by the positionig of cells in the lower row. This is, in software engineering terms, a case of coupling.

The second problem regards the resizing behavior. The total order of tabstops introduces an undesired limitation because unrelated borders of cells in different rows cannot slide past each other. To state the problem in terms of *colspan*: if one border in one row would slide past one in the next row, then certain colspan attributes would have to change. Since these attributes are not changed during resizing, this sliding behavior is not possible during simpe resizing. This restriction can have a negative impact on the space efficiency and visual appearance of a GUI. It may result in unnecessary gaps because a tabstop may be forced to be to the left or to the right of another tabstop even if this does not reflect the structure of the content in the GUI.

The same problem as we encounter in HTML is also found in the Java gridbag layout. The situation here is even worse: all non-simple cells can be placed with reference to the absolute row and column number. This aggravates the problem of interdependency of different parts of the GUI because the parameters for every cell depend on the content between this cell and the origin, in both dimensions.

ALM specifications of areas do not contain such dependencies. In contrast to colspan and rowspan, the concept of areas in ALM typically results only in a partial order of tabstops. This removes the two aforementioned problems of the grid-based approaches: firstly, it leaves more flexibility for the layout engine. Secondly, ALM area specifications are easier to maintain, as they have lower coupling.

8 Related Work

Modularity is recognized as a crucial factor for managing the complexity of software development, and has thus been discussed a lot in the literature, an early seminal work being [17]. One of the questions is that about how to decompose a system into modules, as this is not always obvious from the start. Tools and techniques that facilitate decomposition of artifacts can make a big difference, making management of dependencies and separation of concerns much easier. ALM facilitates such decomposition by using constraints.

Several research projects have used various constraint solving techniques for layout, e.g. [6, 4, 10, 19], with linear programming being one of the most popular techniques, e.g. [16, 1, 7, 9, 3]. These approaches can in principle be applied to GUI layout, athough such approaches often rather targeted constraint-based interactive graphics as they are used in the visulalization of mathematical theorems. ALM is however the first constraint-based layout manager that is comparable with other GUI layout approaches and that directly extends the widespread grid-based layout approaches. ALM differs considerably from these earlier constraint-based approaches in the way it structures layouts with areas and other higher-level constraints. This paper in particular illuminates the modularity issues of GUI layout and how they can be managed using constraints, which is something that has not been discussed before.

ALM is a further development of our earlier tabstop-based layout manager [15], which is in our terminology a first Auckland-style layout manager. In this first layout manager, ordinal and linear constraints were represented separately, and a hybrid algorithm combining topological graph sorting and Gauss-Seidel linear constraint solving was used for layout calculation. For example, the layout manager did not support linear inequalities.

Most GUIs are hard-coded, i.e. represented as source code, and the most popular GUI development tools such as Visual Editor for Eclipse and Visual Studio Designer use a source code representation. It is possible to reverse engineer hard-coded GUIs so that a higher-level ALM specification is recovered [13]. This makes it possible to use ALM with existing GUIs and to leverage existing tools for GUI design. Recovered ALM specifications can be beautified automatically and refined by the developer.

9 Conclusion

This paper introduces the Auckland Layout Model (ALM), which is a cross-platform layout manager based on precise constraint specification. Besides other advantages over common approaches for GUI layout, such as more flexibility when specifying the order of graphical elements, ALM has properties that make it particularly suitable for efficient modular specification and reuse:

- ALM is based on constraints which makes it inherently compositional, enabling separation of concerns into different modules that can be managed separately and flexibly recombined later.

- Developers need only specify what is necessary, i.e. even if a specification is incomplete the solver will come up with a layout, unused features can be ignored, and sensible default values are filled in automatically where necessary.

- Higher-order constructs for areas, rows and columns make it possible to specify GUI layouts on the right level of abstraction, and manage parts of a specification in modular units.

- ALM is based on linear programming and thus allows for the use of standard solver packages. This fosters comparable and reliable performance across platforms.

- ALM offers concrete software engineering advantages over gridbag layouts. It reduces coupling between independent parts of the GUI.

We are using ALM for our own projects, and found it more convenient to use than common layout managers. In particular, it was more efficient and easier to use for complex layouts that would otherwise have required a strongly recursive and deep containment hierarchy. ALM is freely available and can be downloaded at `http://www.cs.auckland.ac.nz/~lutteroth/projects/alm/`.

References

[1] G. J. Badros, A. Borning, and P. J. Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Trans. Comput.-Hum. Interact.*, 8(4):267–306, 2001.

[2] M. Berkelaar, P. Notebaert, and K. Eikland. lp_solve: (Mixed Integer) Linear Programming Problem Solver. 2007. http://lpsolve.sourceforge.net/.

[3] T. Bill, B. Lundell, J. McDonald, and M. Sannella. Bricklayer: window layout using linear programming. Technical report, University of Washington, May 1992.

[4] A. Borning and B. Freeman-Benson. Ultraviolet: A Constraint Satisfaction Algorithm for Interactive Graphics. *Constraints*, 3(1):9–32, 1998.

[5] A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierarchies. *Lisp Symb. Comput.*, 5(3):223–270, 1992.

[6] A. Borning, R. Lin, and K. Marriott. Constraint-based document layout for the Web. *Multimedia Systems*, 8(3):177–189, 2000.

[7] A. Borning, K. Marriott, P. Stuckey, and Y. Xiao. Solving linear arithmetic constraints for user interface applications. In *UIST '97: Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology*, pages 87–96. ACM Press, 1997.

[8] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ, 1963.

[9] H. Hosobe. A scalable linear constraint solver for user interface construction. In *CP 2000: Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming*, pages 218–232. Springer, 2000.

[10] H. Hosobe. A modular geometric constraint solver for user interface applications. In *UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 91–100. ACM Press, 2001.

[11] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, December 1984.

[12] S. Lok and S. Feiner. A survey of automated layout techniques for information presentations. 2001.

[13] C. Lutteroth. Automated reverse engineering of hard-coded gui layouts. In *AUIC '08: Proceedings of the 9th Australasian User Interface Conference*. Australian Computer Society, to appear.

[14] C. Lutteroth, R. Strandh, and G. Weber. Optimal gui layout as a problem of linear programming. Technical Report UoA-SE-2007-6, Software Engineering, The University of Auckland, August 2007.

[15] C. Lutteroth and G. Weber. User interface layout with ordinal and linear constraints. In *AUIC '06: Proceedings of the 7th Australasian User Interface Conference*, pages 53–60, Darlinghurst, Australia, Australia, 2006. Australian Computer Society.

[16] K. Marriott and S. S. Chok. Qoca: A constraint solving toolkit for interactive graphical applications. *Constraints*, 7(3-4):229–254, 2002.

[17] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[18] D. Raggett. RFC1942: HTML Tables, 1996.

[19] M. Sannella. Skyblue: a multi-way local propagation constraint solver for user interface construction. In *UIST '94: Proceedings of the 7th annual ACM symposium on User interface software and technology*, pages 137–146. ACM Press, 1994.

[20] A. Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1986.