Domain Specific High-Level Constraints for User Interface Layout

Christof Lutteroth
The University of Auckland

Robert Strandh
Université Bordeaux

Gerald Weber
The University of Auckland

January 29, 2008

Abstract

We present the Auckland Layout Model (ALM), a constraint-based technique for specifying 2D layout as it is used for arranging the controls in a GUI. Most GUI frameworks offer layout managers that are basically adjustable tables; often adjacent table cells can be merged. In the ALM, the focus switches from the table cells to vertical and horizontal tabulators between the cells. On the lowest level of abstraction, the model applies linear constraints, and an optimal layout is calculated using linear programming. However, bare linear programming makes layout specification cumbersome and unintuitive, especially for GUI domain experts who are often not used to such mathematical formalisms. In order to improve the usability of the model, ALM offers several other layers of abstraction that make it possible to define common GUI layout more easily. In the domain of user interfaces it is important that specifications are not over-constrained, therefore ALM introduces soft constraints, which are automatically translated to appropriate hard linear constraints and terms in the objective function. GUIs are usually composed of rectangular areas containing controls, therefore ALM offers an abstraction for such areas. Dynamic resizing behavior is very important for GUIs, hence areas have domain-specific parameters specifying their minimum, maximum and preferred sizes. From such definitions, hard and soft constraints are automatically derived. A third level of abstraction allows designers to arrange GUIs in a tabular fashion, using abstractions for columns and rows, which offer additional parameters for ordering and alignment. Row and column definitions are used to automatically generate definitions from lower levels of abstraction, such as hard and soft constraints and areas. Specifications from all levels of abstraction can be consistently combined, offering GUI developers a rich set of tools that is much closer to their needs than pure linear constraints. Incremental computation of solutions makes constraint solving fast enough for near real-time use.

1 Introduction

Today many computer programs have graphical user interfaces (GUIs). The developers of such programs have to make sure that the locations and sizes of graphical controls exploit the available screen space efficiently, and contribute to a good usability – a problem known as user interface layout. There are many development tools that support the creation of static GUI layouts, i.e. GUIs in which the locations and sizes of controls remain the same during runtime. However, good GUIs are dynamic, i.e. they can adapt themselves to changes such as resizing of the windows in which the GUI is kept, or different space requirements within the GUI, e.g. due to changing information content. In order to describe dynamic GUI layout appropriately, a developer has to specify its invariants, i.e. the constraints that remain valid during runtime while the actual locations and sizes may change.

Unfortunately, there is no standard way of describing GUI layout constraints. Since there are many different computer architectures, operating systems, programming languages and development frameworks, numerous different GUI technologies have evolved. Each of these technologies has their own quirks and requires a specific know-how, particularly when it comes to GUI layout. As a result, it is important for research in that area to abstract from individual technologies and consider GUI layout from a more formal perspective, i.e. as a constraint solving problem. Several research projects have used various constraint solving techniques for GUI layout, e.g. [7, 6, 17, 27], with linear programming being one of the most popular techniques, e.g. [24, 2, 9, 15, 5].

At the same time one has to keep in mind that developers and GUI designers need tools that are easy to comprehend, i.e. which can be used efficiently without too much formal background. A problem with formal constraint solving techniques is that they are cumbersome to use when applied to domains such as GUI layout. Their concepts are usually very generic and thus on a very low level. This leads us to the necessity of higher-level constraints for GUI layout, i.e. abstractions that are tailored to the needs of this domain and easily understood by its practitioners.

In this paper we present the Auckland Layout Model (ALM), which is a higher level constraint language that captures natural abstractions for the domain of GUI layout. The focus is on 2D layout and the concepts are very general, so that they can be applied to print layout of documents as well as to the design of resizable, window-based GUIs. The language is an important part of a larger project, the Auckland Layout Manager, which is a platform independent library for GUI layout with concepts that overcome some of the problems of common GUI layout frameworks. The Auckland Layout Manager implements the constraint language presented here in the shape of an API, and was successfully applied in larger software packages such as a software engineering tool suite. ALM greatly simplified the GUI development of these projects.

General information about ALM and its implementation is given in Sect. 2. Section 3 explains the significance and use of linear constraints for GUI layout. ALM offers different types of abstractions that can be grouped into layers. Having several levels of abstractions can fulfill different purposes; it can be used to simplify the constraints solving process [14] or it can make the creation of models easier. We focus here on the second purpose. The concept of rectangular areas is very important for layouts, therefore ALM offers abstractions to support this concept, which are described in Sect. 5. Areas have certain classic parameters specifying their minimum, maximum and preferred sizes. Rectangular areas are often grouped into rows and columns, therefore there are other abstractions that make these concepts first class for certain kinds of constraints. Abstractions for rows and columns are described in Sect. 6. We will discuss how higher levels of abstraction are translated to lower levels of abstraction, and how a specification is eventually used to generate an instance of the linear programming problem. Section 7 presents some small layout examples. ALM's abstractions can be used to conveniently specify typical layout patterns, and in fact, a pattern approach is very much applicable and important for this domain. This is discussed in Sect. 8. Section 9 discusses related work, and Sect. 10 evaluates the good and bad points of ALM, particularly pointing out its shortcomings and limitations. Section 11 concludes the paper.

2 The Auckland Layout Model

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 every control manually, and also 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 in that window. Many modern GUI toolkits include layout engines, which support developers in that task. 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 can then position and size the controls according to some mechanism.

A modern toolkit usually contains several different layout engines, some only supporting very simplistic layouts and others more sophisticated ones. A row layout would, for example, simply 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. Often a developer can, for example, specify upper and lower limits for the size of each control, and also 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.

The ALM presented in this paper was implemented in the Auckland Layout Manager, which is a layout engine that allows it to specify GUI layout formally using the ALM. Like most good layout engines, the Auckland Layout Manager arranges the elements of a GUI in a tabular fashion. However, it allows for more flexibility of the layout by allowing GUI designers to leave the order of the elements in the GUI partially undefined. It allows a GUI designer to specify the dynamic behavior of a layout, i.e. the behavior under changing environment conditions such as a reduced window size. Such specifications help a layout engine to optimize the distribution of available screen space. In contrast to many popular layout engines, the Auckland Layout Manager has a solid mathematical foundation. GUI developers can specify a layout on different layers of abstraction, with the more abstract layers being based on the ones below. On the lowest level of abstraction, layouts are specified by linear constraints and a linear objective function. Optimal layouts are calculated using linear programming. This means that all layouts are eventually specified in terms of linear algebra.

The ALM based on linear constraints is a representative 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 models that use basically the same qualitative layout model, especially the tabstop approach and the option to specify only a partial order on tabstops, are called *Auckland-style layout models.*

2.1 General Considerations

User interfaces, just as practically any form of well-structured document, organize their content in rectangular areas, or areas that can be suitably described with a rectangular bounding box. Also the area on which the GUI or document is laid out is usually rectangular itself. By painting the rectangular areas in a particular order, i.e. each on a particular layer, we can reproduce the complete document or GUI. Another important property one can observe is that there exist relationships between the edges of the elements' rectangular areas. Usually, there are particularly important virtual lines spanning across the overall area to which groups of other areas are aligned. The alignment to common edges serves to structure the represented content, e.g. as seen in lines and paragraphs of text, and adds to a clear overall appearance.

When we extend every edge of every rectangular area in the GUI to be either a vertical or horizontal unbounded line, we end up with a grid in which all parts of the GUI are placed. This grid can be described as a generalization of an ordinary table, which is a widely used and intuitively understood graphical artifact. This common and versatile structure can serve as a basis for structured graphical content, and that is why our methodology is based on a tabular layout.

The ALM can be understood as a generalization of the table construct as it is present in many document markup languages. To the practitioner, tables are known as a very general layout tool; in HTML, for example, tables are often used as the main means to create desired page layouts (although they are being more and more replaced by Cascading Style Sheets). In those approaches, tables are directly defined as rows and columns. In the ALM, the focus switches to the borders between the table elements, which are conceived as vertical and hori-

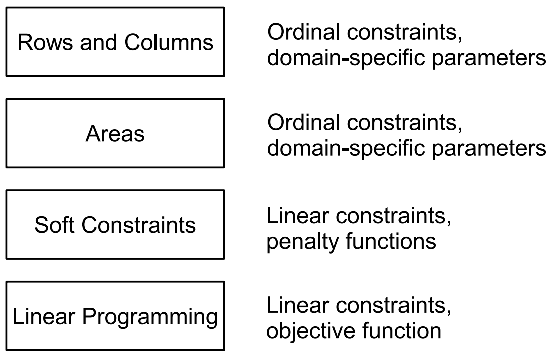

Rows and Columns
Ordinal constraints, domain-specific parameters
Areas
Ordinal constraints, domain-specific parameters
Soft Constraints
Linear constraints, penalty functions
Linear Programming
Linear constraints, objective function

Figure 1: The different levels of abstraction offered by the ALM.

zontal tabulators and called *x-tabstops* and *y-tabstops*, respectively. Together they are simply called tabstops, or *tabs* for short. Tabstops are symbolic entities with a name, and correspond to the variables of the specification. Solving the specification means assigning each x-tab an x-coordinate and each y-tab a y-coordinate in the coordinate system of the GUI.

2.2 Levels of Abstraction

As pointed out, ALM offers different levels of abstraction, which are illustrated in Fig. 1. On the lowest level, a layout specification is a linear programming problem. Linear programming is an optimization technique that minimizes (or maximizes) an *objective function* subject to a set of *constraints*. In linear programming, the objective function is a linear combination of the *variables*:

$$c_1x_1 + c_2x_2 + \ldots + c_nx_n$$

where each x_i is a variable and each c_i is a constant coefficient. The constraints are linear equalities and inequalities. In other words, linear programming is about finding values for all variables $x_1, \ldots, x_n$ so that all constraints are satisfied and the objective function value is either as large or as small as possible. Several software packages (commercial and open-source) exist for solving linear programming problems of this type. Most of them use the well-known *simplex method* [11] which, although its worst-case asymptotic complexity is exponential, is known to be very fast in practice. The Auckland Layout Manager uses the lp_solve package [4], which is a freely available, open-source linear optimizer.

Over-constrained specifications are a common problem when specifying GUI layout with constraints. In order to keep the specification feasible, soft constraints can be used, which only add a penalty to the objective function if violated. The layer on top of the linear programming solver manages soft constraints for the developer, which are transparently translated into hard constraints and terms in the objective function. ALM supports soft constraints with complex penalty functions that allow GUI developers to specify variables with affinities of varying strengths for several particular values.

The next layer of abstraction provides functionality for managing rectangular areas of the GUI, which may contain controls or other graphical elements. This layer allows developers to specify the topology of the elements in the GUI using ordinal constraints, and offers domain-specific parameters such as size constraints for an area. The layer for areas builds on the layer for soft constraints.

The top layer offers abstractions for rows and columns in which areas can be aligned, similar to common spreadsheet applications. This layer employs ordinal constraints to define the order of the rows and columns, and offers domain-specific parameters for the alignment of areas in table cells. All underlying layers are used in order to translate rows and columns into low-level specifications.

2.3 Performance

The Auckland Layout Manager has been subjected to several performance tests, and has also been used in real applications with rich and complex GUIs. Our experience has shown that large feasible layout specifications can be efficiently solved on modern hardware within milliseconds, and that the performance is good enough for near real-time use such as immediate response to dynamic resizing of a GUI with the mouse. Infeasible specifications could take significantly longer to process than feasible specifications, but once a specification was proven feasible, processing time had a reasonable bound.

Like other linear programming based systems used for GUI layout such as [24, 2, 15], the Auckland Layout Manager uses an incremental approach. Layout specifications are constructed and modified incrementally: if a specification is changed by a GUI developer, corresponding changes of the internal representation are kept to a minimum. Similarly, solutions are calculated incrementally by using a previous solution as initial value. Layout specifications frequently contain redundant information, which is automatically removed before starting the optimization process. All this reduces constraint solving time drastically.

We have tested our implementation of ALM 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. 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 left layout uses three areas, as described in Sect. 5, and two linear constraints, as described in Sect. 3. The right layout uses the abstractions for rows presented in Sect. 6, with different alignment settings for each of the three rows. The more sophisticated layout in Fig. 3 contains 14 areas, using different settings for alignment and margins. It takes about 0.7 milliseconds on a Pentium M with 1.6 GHz, and about 1.4 milliseconds on a Pentium 3 with 788 MHz.

The layout illustrated in Fig. 4 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

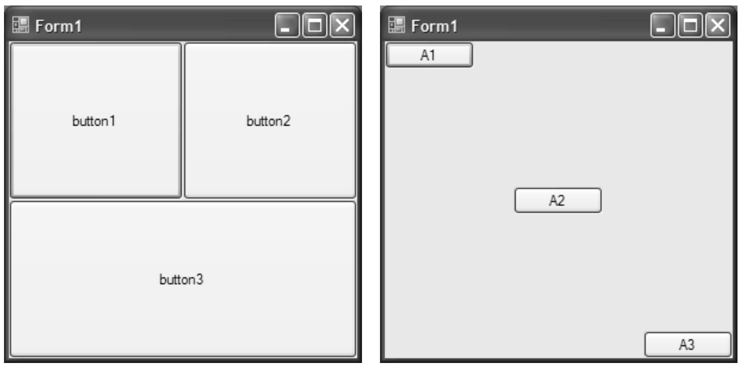


Figure 2: Left: Layout with three areas and two additional constraints. Right: Buttons aligned in three equally high rows, i.e. top-left aligned, centered, and bottom-right aligned.

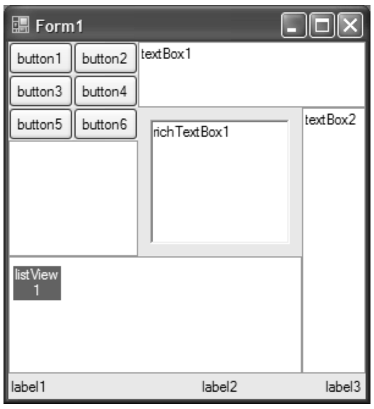


Figure 3: Layout with 14 areas.

others. On a Pentium M with 1.6 GHz the layout calculation takes about 6 milliseconds, and on a Pentium 3 with 788 MHz it takes about 15 milliseconds.

3 Specifying GUI Layout with Linear Constraints

Like in many other approaches, layout specification of ALM is based on linear programming, which makes use of linear constraints. Tabstops are used as the constraint variables and are interpreted as x- and y-coordinates in the layout. So 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 the set of possible constraints on them looks as follows:

$$\begin{aligned} \{ \quad & a_0x_0 + \ldots + a_mx_m + b_0y_0 + \ldots + b_ny_n \; OP \; c \\ | \quad & a_0, \ldots, a_m, b_0, \ldots, b_n, c \in \mathbb{R} \; \wedge \; OP \in \{\leq, =, \geq\}\} \end{aligned}$$

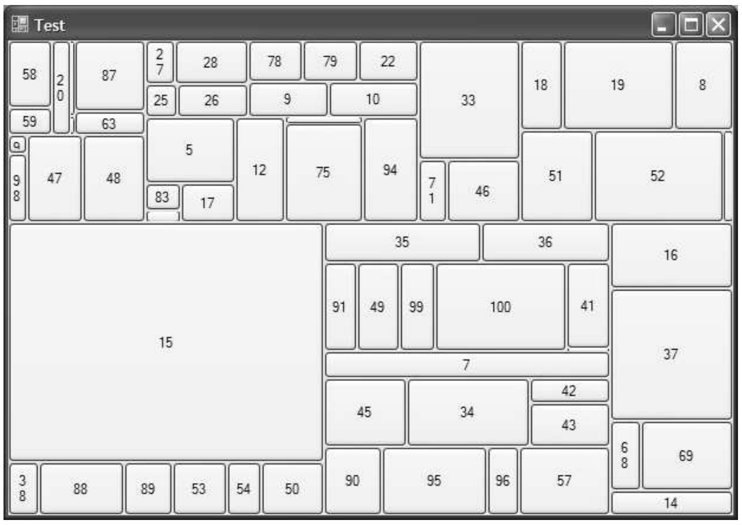


Figure 4: Randomly generated layout with 100 areas.

Note that 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 such as cm, which always produces the same size. For this to work, all values are transparently converted to pixels, using the properties of the active display. 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 quite easily.

3.1 Absolute Constraints

We use absolute constraints in order to place x- or y-tabstops at particular x- or x-positions of the GUI, or set the width or height between tabstops to a fixed value. For example, if we want 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 GUI, i.e. the properties that do not change when, e.g., resizing the window the GUI is displayed in. Note that absolute constraints may be impossible to satisfy under some circumstances. For example, if the available display area is only 10cm wide, the width between two x-tabstops cannot exceed this value.

3.2 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, such as GUI display size or resolution changes. The layout manager recalculates the layout when such a change occurs.

Relative constraints can be used in order to position tabstops at positions relative to other tabstops. For example, one might 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. For example, if we want the width between x-tabstops x_1 and x_2 to be twice as much as 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. For example, we could specify the aspect ratio for an area $(x_1, y_1, x_2, y_2, \mathit{moviepanel})$, 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 set the ratio of width and height of this area to a standardized ratio such as 16:9. This would be achieved with the following constraint:

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

4 Soft Constraints

The linear constraints discussed in the last section are hard, i.e. if they are contained in the specification of a layout, then this layout will satisfy them strictly. However, sometimes we want to specify constraints that may not be satisfied fully if circumstances do not permit so. Such constraints are called *soft constraints*, and are a common technique for dealing with over-constrained problems [25, 16]. They are natural in applications for user interfaces and have been used in the past [2].

Let us consider an example. We have four x-tabstops, $x_1 \leq x_2 \leq x_3 \leq x_4$, and the following hard constraints:

$$C = \{x_2 - x_1 = 2(x_4 - x_3), x_4 - x_3 \geq 50\}.$$

According to the constraints the width between x_1 and x_2 has to be twice as large as the width between x_3 and x_4, which in turn must not be smaller than 50. If we have a total space available on the horizontal of less than 150, then these constraints cannot be satisfied. However, we might want to be able to render a layout for total widths less than 150, and we might decide that the first constraint is not vital under such circumstances. While we want the constraint to be satisfied whenever possible, we might permit a little deviation as much as necessary. The solution is to transform the constraint into a soft one.

Soft constraints are basically ordinary linear constraints together with one or two penalty coefficients. Soft linear equalities have two penalty coefficients, p_{pos} and p_{neg}, which describe how much deviations from an exact solution of the constraints in a positive or negative direction are penalized. If one does not need to distinguish positive and negative deviations, but wants to penalize deviation in all directions in the same manner, the penalties are set so that $p_{pos} = p_{neg}$. Soft linear inequalities need only a single penalty coefficient p because inequalities using the $\leq$ operator can only have positive deviations, and inequalities using the $\geq$ operator can only have negative deviations. The higher the penalty coefficients the more will the layout engine favor the soft constraint over others, trying to minimize the overall penalty, i.e. soft constraints can be prioritized. More about this will be described later on.

Soft constraints are important because in contrast to hard constraints they guarantee that there is always a solution. If we add hard constraints to a layout specification the space of layout solutions can easily turn out to be empty under certain circumstances, or even empty altogether. Hence several approaches for prioritizing constraints have been devised, including constraint hierarchies [8]. It is a well known phenomenon that natural specification approaches for user interfaces can lead to over-constrained problem definitions, if the constraints would all be considered as non-negotiable. This can happen accidentally, e.g. when different GUI developers add their own constraints without being aware of the possibility of making the layout infeasible.

Therefore, it might even be advisable to use only soft constraints on a certain level, in order to avoid corrupt specifications that do not have a solution under all possible circumstances. The idea is similar to that of superuser and user modes in operating systems: in the superuser mode, program code is capable of crashing the system and therefore has to be developed carefully. In user mode, crashing the system should not be possible, therefore the capabilities are restricted. Analogously, it might be advisable to allow only soft constraints for regular GUI development. By giving them high enough penalty coefficients, soft constraints can be made to behave like hard ones, while guaranteeing that they can be violated if there would be no solution otherwise. The choice of the penalty coefficients will be discussed later in Section 4.7.

With soft constraints it is possible to simulate a spring-like behavior between tabstops while avoiding conflicts with other constraints. Springs basically cause the sizes of a layout to grow and shrink proportionally to spring constants, which are properties assigned to individual widths and heights. This basically results in linear constraints with the proportions taken up by individual areas determined by the coefficients, as discussed in Sect. 3.2.

4.1 Representing Soft Constraints

Since linear constraints are an intrinsic part of the linear programming problem, all hard linear constraints can directly be handed over to the optimizer just as they are. Soft constraints require a bit of extra work and are reduced to several hard linear constraints and a summand of the objective function, i.e. using a weighted cost model. We chose to implement soft constraints on top of normal linear programming since many good solvers do not support soft constraints explicitly. There are solvers such as Cassowary [2] that support soft constraints but do not offer the numerical representation of penalties chosen for ALM. Such a representation arises naturally from the way soft constraints are represented, as described in the following. As mentioned before, a linear inequality that is treated as a soft constraint has a single penalty coefficient p. Linear equations that are soft constraints can be treated as two linear inequalities with operators $\leq$ and $\geq$, respectively. The penalty coefficient p_{pos} of the soft linear equality corresponds to the penalty coefficient of the $\leq$ inequality, and the penalty coefficient p_{neg} corresponds to the penalty coefficient of the $\geq$ inequality. Consequently, it is enough to know how to represent soft inequalities.

For each soft constraint of the form

$$a_0x_0 + \ldots + a_mx_m + b_0y_0 + \ldots + b_ny_n \ \leq_{soft}\ c \tag{1}$$

and a penalty coefficient p, we hand a hard constraint of the following form to the optimizer:

$$a_0x_0 + \ldots + a_mx_m + b_0y_0 + \ldots + b_ny_n - \delta \ \leq\ c \tag{2}$$

with δ being a new variable in our system. The variable δ indicates how much the soft constraint is violated and is called the *deviation variable* of this soft inequality. If the inequality (1) is to be read as a hard constraint, we call this the hard constraint *corresponding* to the soft constraint. For the soft constraint, we use a subscript *soft* at the operator for clarification. We call the hard constraint (2) handed down to the optimizer the hard constraint *underlying* the soft constraint.

We pose that $\delta \geq 0$, and want $\delta > 0$ only if the left side of the soft constraint needs indeed to be greater than the constant on the right side, i.e. if a violation occurs. With just the linear constraints this is not guaranteed: an arbitrarily large constant can be added to δ without changing the solution of the aforementioned constraints. In order to solve this problem we have to add a summand to the objective function that has the form $p\delta$ with $p > 0$. Because the optimizer

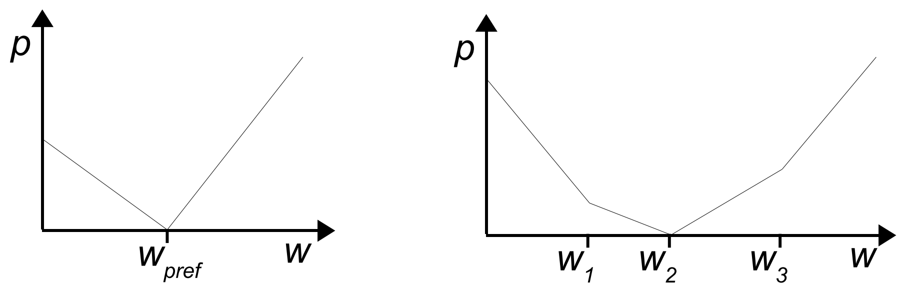


Figure 5: Examples for functions of area width over penalty. Left: example penalty function as defined by the penalty parameters of an area. Right: example of an arbitrary convex penalty function.

will minimize the objective function, δ will be chosen no greater than necessary. Soft inequalities with operator $\geq$ are handled analogously.

For linear equations, according to the aforementioned translation into two inequalities, two variables $\delta_{neg} \geq 0$ and $\delta_{pos} \geq 0$ are introduced. Because the optimizer will minimize the objective function, both δ_{neg} and δ_{pos} will be chosen no greater than necessary, which means that at least one of the new variables will be zero. The optimal solution of the system will tend to allow more deviation for soft constraints with small penalty coefficients, as they contribute less to the overall penalty, and allow less deviation for soft constraints with high penalty coefficients, as they contribute more. This is exactly the behavior that we want.

4.2 Arbitrary Convex Penalty Functions

With the penalty coefficients described in the last section, positive resp. negative deviations from a preferred value are penalized with a single penalty coefficient. For example, if we look at how the penalty p changes with the width w of an area for which a preferred width w_{pref} has been set in a soft constraint, we see that it generally looks like the left side of Fig. 5. We call this a *penalty function.* The penalty is zero for $w = w_{pref}$ and behaves linearly for $w < w_{pref}$ and $w > w_{pref}$. Usually the penalty coefficients are greater than zero, so that we have a dropping line ending at $(w_{pref}, 0)$ to the left of w_{pref}, and a rising line stretching out from it on the right.

In some rare cases one might desire a more sophisticated penalty function, such as the one depicted on the right side of Fig. 5 for the width of an area. Such a penalty function can be arbitrarily composed of linear segments, i.e. it can be a piecewise linear function, with the restriction that the resulting function is convex. With such penalty functions deviations from a preferred value can be penalized in a nonlinear fashion, i.e. larger deviations can be penalized over-proportionally stronger than smaller deviations. Like this, nonlinear penalty functions such as quadratic functions can be approximated.

Arbitrary convex penalty functions can be defined with the following parameters:

- a set of sample points $(w_1, p_1), \ldots, (w_n, p_n)$, through which the function runs,
- a penalty for $w = 0$,
- a coefficient describing the gradient of the function for $w > w_n$.

Without loss of generality, this definition refers to a function between width and penalty. Other penalty functions can be defined in the same manner.

The concept of arbitrary convex penalty functions in ALM is a higher-order construct that can, like all ALM features, ultimately be reduced to the problem of linear programming. How this reduction takes place will be discussed in the next section.

4.3 Representing Arbitrary Convex Penalty Functions

In this section we want to describe how arbitrary convex penalty functions can be approximated on a finite interval in the form of additional linear constraints and an additional summand of the objective function. Without loss of generality, let w be the variable that we want to define the penalty function on. As described in Sect. 4.2, arbitrary convex penalty functions can be linearly approximated in as much detail as necessary with a set $(0, p_0), (w_1, p_1), \ldots, (w_n, p_n)$ of $n + 1$ sample points, with $0 < w_1 < \ldots < w_n$, and a coefficient $c > 0$ describing the gradient of the function for $w > w_n$. A penalty function could theoretically be defined on $\mathbb{R}$, but we limit this discussion to $\mathbb{R}_0^+$ without loss of generality.

For each w_i, $i \in \{1, \ldots, n\}$, we define a soft constraint $w = w_i$ with penalty coefficients p_{neg_i} and p_{pos_i}. This means that we add hard constraints to the system of the form

$$w + \delta_{neg_i} - \delta_{pos_i} = w_i,\ \ \delta_{neg_i} \geq 0,\ \ \delta_{pos_i} \geq 0$$

and add summands to the objective function that have the form

$$p_{neg_i}\delta_{neg_i} + p_{pos_i}\delta_{pos_i}.$$

Now we can choose the penalty coefficients p_{neg_i} and p_{pos_i} in a way that those summands added to the objective function behave, as a whole, like the penalty function we want to represent.

At this point it is important to understand why only convex penalty functions can be specified. The reason for this is that each soft constraint must have penalty coefficients p_{neg_i} and p_{pos_i} so that

$$p_{neg_i} + p_{pos_i} \geq 0.$$

This constraint is important in order to keep the linear programming problem bounded, i.e. constrained so that the value of the penalty function cannot be made arbitrarily small. If the penalty coefficients add up to a negative value, then δ_{neg_i} and δ_{pos_i} could be made arbitrarily large, canceling out each other in the soft constraint and making the value of the objective function arbitrarily small. The problem would be unbounded. This means that a negative p_{neg_i} requires a positive p_{pos_i} at least as large as the absolute of p_{neg_i}, resulting in a straight line or an upward bent. Analogously, a negative p_{pos_i} requires a positive p_{neg_i} at least as large as the absolute of p_{pos_i}, again resulting in a straight line or an upward bent. This monotonic increase of the gradient of each partial penalty function causes the total penalty function to be convex as well.

In order to understand how to choose the penalty coefficients, let us have a closer look at the term that is added to the objective function. First of all, the individual summands that this term is made up of are all continuous because they consist, as described in Sect. 4.2, of two linear functions that meet at one point. As a consequence, the term we want to look at, which is the sum of those summands, is continuous as well. In general, this term looks like this:

$$p_{neg_1}\delta_{neg_1} + p_{pos_1}\delta_{pos_1} + \ldots + p_{neg_n}\delta_{neg_n} + p_{pos_n}\delta_{pos_n}.$$

For $0 \leq w \leq w_1$, $\delta_{neg_i} \geq 0$ and $\delta_{pos_i} = 0$ for all $i \in \{1, \ldots, n\}$ because w is smaller than all the w_i. Consequently, the term of the objective function looks like this:

$$\begin{aligned} & p_{neg_1}\delta_{neg_1} + \ldots + p_{neg_n}\delta_{neg_n} \\ = \; & p_{neg_1}(w_1 - w) + \ldots + p_{neg_n}(w_n - w) \\ = \; & (p_{neg_1}w_1 + \ldots + p_{neg_n}w_n) + (-p_{neg_1} - \ldots - p_{neg_n})w \end{aligned}$$

On this interval, we want the term to behave like a linear function going through the points $(0, p_0)$ and (w_1, p_1), i.e. like the term

$$p_0 + \frac{p_1 - p_0}{w_1}w.$$

In order to realize this, we choose $p_{neg_1}, \ldots, p_{neg_n}$ so that the following system of linear equations is satisfied:

$$\begin{aligned} p_{neg_1}w_1 + \ldots + p_{neg_n}w_n &= p_0 \\ -p_{neg_1} - \ldots - p_{neg_n} &= \frac{p_1 - p_0}{w_1} \end{aligned}$$

Note that this system has exactly one solution for $n = 2$, and an unlimited number of solutions for $n > 2$. However, as we have seen before this does not mean that we can actually choose every such solution. Convexity has to be preserved to keep the problem bounded. If we choose to set a penalty coefficient to zero, this means we can replace the respective equality soft constraint by an inequality soft constraint since only deviations to one side are penalized.

For $w_j \leq w \leq w_{j+1}$, $j \in \{1, \ldots, n-1\}$, $\delta_{neg_i} = 0$ and $\delta_{pos_i} \geq 0$ for $i \in \{1, \ldots, j\}$. Furthermore, $\delta_{neg_i} \geq 0$ and $\delta_{pos_i} = 0$ for $i \in \{j+1, \ldots, n\}$. The term of the objective function looks like this:

$$\begin{aligned}
& p_{pos_1}\delta_{pos_1} + \ldots + p_{pos_j}\delta_{pos_j} + p_{neg_{j+1}}\delta_{neg_{j+1}} + \ldots + p_{neg_n}\delta_{neg_n} \\
= \; & p_{pos_1}(w - w_1) + \ldots + p_{pos_j}(w - w_j) \\
& + p_{neg_{j+1}}(w_{j+1} - w) + \ldots + p_{neg_n}(w_n - w) \\
= \; & (-p_{pos_1}w_1 - \ldots - p_{pos_j}w_j + p_{neg_{j+1}}w_{j+1} + \ldots + p_{neg_n}w_n) \\
& + (p_{pos_1} + \ldots + p_{pos_j} - p_{neg_{j+1}} - \ldots - p_{neg_n})w
\end{aligned}$$

On this interval, we want the term to behave like a linear function going through the points (w_j, p_j) and (w_{j+1}, p_{j+1}), and we do this by choosing p_{pos_j}. Note that all $p_{neg_1}, \ldots, p_{neg_n}$ have already been chosen to configure the function for $0 \leq w \leq w_1$. Furthermore, $p_{pos_1}, \ldots, p_{pos_{j-1}}$ are already used for configuring the function in $0 \leq w \leq w_j$. We only need to configure one parameter because we only need to set the gradient of the linear function so that it passes through (w_{j+1}, p_{j+1}) if it has passed through (w_j, p_j) before. We can be sure that the linear function passes through (w_j, p_j) because the previous linear segment was configured so that it passed through that point, and the overall function is continuous. So all we need to do is choose p_{pos_j} so that

$$p_{pos_1} + \ldots + p_{pos_j} - p_{neg_{j+1}} - \ldots - p_{neg_n} = \frac{p_{j+1} - p_j}{w_{j+1} - w_j},$$

which has exactly one solution.

For $w \geq w_n$, $\delta_{neg_i} = 0$ and $\delta_{pos_i} \geq 0$ for $i \in \{1, \ldots, n\}$. The term of the objective function looks like this:

$$-p_{pos_1}w_1 - \ldots - p_{pos_n}w_n + (p_{pos_1} + \ldots + p_{pos_n})w.$$

Values for all free parameters except p_{pos_n} have been chosen already, so all we need to do now is choose p_{pos_n} so that

$$p_{pos_1} + \ldots + p_{pos_n} = c.$$

The whole process of determining the p_{neg_i} and p_{pos_i} can be done beforehand and has no footprint on the performance of the actual layout optimization. This process can be completely hidden from the user's view of the system, so that GUI developers can specify arbitrary convex penalty functions only with the parameters listed in Sect. 4.2, and without any understanding about the actual implementation.

4.4 Constants as Abstractions

An inconspicuous, but indeed important concept of abstraction is to compute, wherever possible, the coefficients in the linear program from symbolically defined constants. An example would be the paper size in a print layout. If we want to modify the problem so that a scaled layout for a different paper size is produced, then certain coefficients in the linear program have to be changed. If the model is not defined in such a way that the constants that need to change (typically by being scaled by a certain ratio) are clearly distinguished, then the problem definition is not maintainable – a clear indication of a lack of abstraction. If all dependent coefficients are made explicit by using the paper size as a symbolic constant, this gives a maintainable model: in the case of a change in this constant, possibly all constraints have to be redefined. If a constraint solver is used in a third generation language in the same manner as it is done with the Auckland Layout Manager, then these constants are not necessarily also constants in the third generation program; if the constants are changed in the running program, a re-execution of the code that is defining the problem can be performed. In the future, it might become more and more common practice not to hard-code the constraint generation; instead, the constraints might be given as documents, i.e. as structured data that can be changed and processed at runtime.

4.5 Default Constraints

Default constraints should provide an expected or sensible behavior in cases where the user has not specified a preferred behavior. A simple example is a layout consisting of two areas next to each other. If the programmer does not specify the mutual width, then the natural assumption would be that the areas are rendered with equal width. With respect to the model, default constraints should specify a unique solution if the constraints specified by the user do not yet give a unique solution. Default constraints can be overridden by user specifications in two ways. Either, the default constraint is only added if no user constraint is present. This requires a symbolic management of the problem specification. Alternatively, default constraints are marginalized by user constraints. For this purpose, default constraints are specified as soft constraints.

4.6 Interactions between Soft Constraints

There is a common misconception about soft constraints in linear programs. Generally, modelers might fear that soft constraints have a residual influence: either harder soft constraints, which should simply be fulfilled, could wiggle out of place a bit (a pixel or two, creating an annoying misalignment), or weaker constraints that are violated for good could still exert influence on the solution. Here the linearity of our model is an advantage, making both cases rather unproblematic. In an optimization problem we can consider the optimal solution to be a function of the direction of the objective function, i.e. the penalty coefficients. For linear programs, this function is uniquely defined almost everywhere. The only unique solutions are the finitely many vertices of the solution space polytope. This function is discontinuous and piecewise constant. If the direction of the objective function becomes perpendicular to at least one of the edges at this vertex, the solution becomes ambiguous. Further rotation makes the opposite vertex at this edge the new solution. Since the solution is always a vertex, only full jumps of the solution from one vertex to another are possible.

4.7 Constraint Hierarchies

The concept of soft constraints first establishes a heterogeneous hierarchy of two constraint types, hard constraints and soft constraints. We want to understand whether we can view both types as being in principle of the same kind. Moreover, in many applications one might want to have even multi-step hierarchies of numerical constraints. Constraint hierarchies are described in [8], and used, for example, in [2, 15]. We discuss here how the notion of soft constraints establishes hierarchies.

A constraint hierarchy is a partial order on soft constraints, so that *softer* constraints will be easier violated in an informal sense. Different penalty coefficients are enabling such a hierarchy of constraints. Consider a layout description with two soft constraints A and B, where B is supposed to be the softer of the two. In order to be softer, constraint B has to have a smaller penalty coefficient than A. Consider further the case that the hard constraints corresponding to A and B are conflicting. In this way, only if A is satisfiable without penalty and the whole description without B has a non-unique solution, then B will have an influence on the solution. Otherwise, B has only an influence on the value of the objective function for the optimal solution.

However, in the presence of many soft constraints care has to be taken in the following sense. GUI developers may have an intuitive notion of several classes of soft constraints that they want to use. Each such class we want to call a *soft constraint level.* A natural approach is to use the same penalty coefficient for each soft constraint in this class. Now, if several soft constraints in one class are to be violated simultaneously, the penalties incurred add up. Thus, if a certain number of related soft constraints are inserted, they might haphazardly become more influential than a different constraint that was supposed to have priority. This will be one reason to introduce the concept of myriads in due course.

Note that one solution that comes to mind would involve a redefinition of soft constraints, more specific, soft inequalities: soft inequalities could be made belonging to the same class by letting them all use the same deviation variable. This would mean that once one constraint in this class requires a certain value of a deviation variable δ, then all constraints using this deviation variable would get the same play or leeway. As a result, more soft constraints on the same level will not add up influence in the way described before. However, such an approach does not lead to the usual behavior we want in soft constraints, where every single constraint should be fulfilled in a best-effort approach. Therefore we do not use such a concept for the soft constraints.

A second problem is that the influence of a soft constraint might be enlarged through other linear dependencies in an effect we call the *pantograph problem.* A *pantograph* in a layout is an informal name we want to give to a situation where the features in one small area have a scale-up copy in some other area, say scaled up by a factor of 10. We consider layouts as modifiable, and these two copies will always change in unison, but with a scale factor of 10. This is similar to the tool called a pantograph that is used in technical drawings. A very simple pantograph is given by a single linear constraint of the form:

$$10(a_{small} - b) - (a_{large} - b) = 0$$

where just the tabstop a_{large} is the scale-up copy of a_{small}. If now there are soft constraints on the scale-up copy, these act also as soft constraints on the smaller copy. However, in absolute coordinates, these soft constraints are 10 times harder on the smaller copy. This is the pantograph problem in determining the mutual hardness of soft constraints.

The GUI developer often wants to ensure that both effects, the adding up of penalties as well as the pantograph problem, do not influence the constraint hierarchy. Formally we want to achieve that no linear combination of the penalty coefficients from one soft constraint level becomes larger than any penalty coefficient from a higher soft constraint level. This is obviously not achievable with penalty coefficient from the field of the other coefficients. Theoretically this problem can be solved elegantly, if the whole linear problem is considered as a problem over a non-Archimedean number field. A natural example of such a field is the set of (fractional) rational functions. Such a field has numbers of different levels in an infinite hierarchy, each level appearing to the next lower level as infinite. We want to use the features of this extension only with respect to penalty coefficients of soft constraints. For all other variables we introduce inequalities that restrict their range to the ordinary real numbers. Also all other coefficients must be real numbers.

Since we do not have non-Archimedean numbers in standard linear solvers, we look for approximations. A straightforward approximation is of course to use coefficients of different orders of magnitude, for example choosing coefficients of level one in the order of unity, but choosing weights in the level with the next higher priority in the order of say 10000. This gives a very workable solution. We now explain and motivate this workaround in analogy to the concept of non-Archimedean numbers.

It is a good pattern to use a first abstraction for the definition of such constraints. The abstraction should encapsulate the choice of the constant that establishes the hierarchy. We call the constant a *myriad* and denote it with capital M, since any confusion with the shorthand for "Mega" would be harmless. The word "myriad" stems from classic Greek "myrioi", a term with two meanings: depending on the pronunciation, it either means the number "tenthousand" or the indefinite numeral "countless"; hence it reflects our use of the myriad as a symbol for a very big number that serves as a replacement for Archimedean infinity. It is a good abstraction to express penalty coefficients as polynomials of the myriad: each level in the hierarchy of constraints is assigned a power of the myriad. Note therefore that in this way our problem definition is not already tied to the decision to "cheat" by using a big finite number for the myriad.

Again, this first abstraction has the disadvantage that it assigns each soft constraint level a fixed priority. An obvious next step is not to put powers of the myriad explicitly into the constraint, but to choose symbolic names for each such level. We call such a constant which acts as a symbolic name a *rank unit*. This term indicates that the purpose of this constant on the one hand is to establish a rank between constraints; on the other hand the term uses an analogy to two physical units describing different physical dimensions. In an example problem, we could have one rank unit `image` that applies to all image dimensions, and one unit `userresize` that applies to all enduser-chosen GUI dimensions. Soft constraints of the rank unit `userresize` are changed, for instance, if the user resizes a frame for a photo album on the screen. If the system is programmed using rank units consistently, then the GUI behavior can be influenced late or even at runtime in the preferences menu of the application: if we choose `image` $= M$`userresize`, and the enduser resizes the frame, then the size changes according to his wishes until the frame would become too small to show all images; below that size, shrinking would not work. If we choose `userresize` $= M$`image`, then the enduser could shrink the frame arbitrarily. We might actually find that in a good definition of a linear layout, all coefficients have a rank unit. In the proposed translation of the myriad into a big finite number, and if the underlying number type are floating point numbers, then the number of levels is actually limited and small, but this is acceptable especially in our application context.

5 Areas

This section describes abstractions for the rectangular areas in the GUI that contain its graphical controls. First we will discuss now how it is possible with these abstractions to specify the topology of the different elements that make up the GUI, without actually giving any quantitative data such as positions or dimensions of controls. We call this the *qualitative layout model* of ALM. Then we will discuss the metric properties of areas.

5.1 Problems of Total Order in Table Specifications

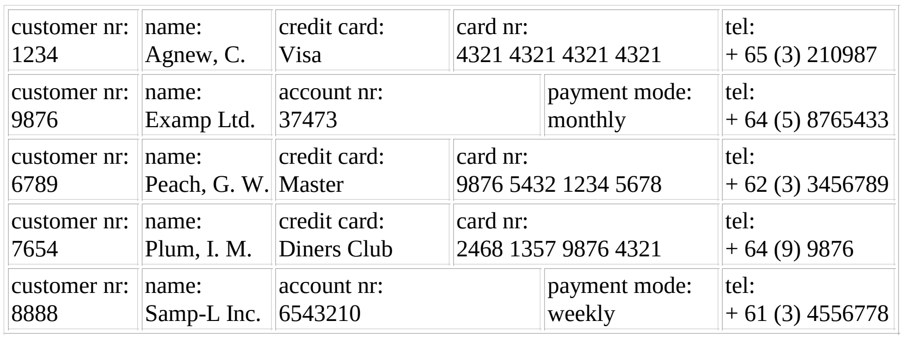
The following example shows why ALM allows partial ordering of tabstops and does not require a total order like most other approaches do. Consider a table that shows objects of two subtypes of a common class. For example, consider a list of customers that contains both private and corporate customers, such as the one in Fig. 6. Both customer types have columns for the customer number and name. Corporate customers have an open account and a payment mode, while private customers have a credit card type and an a credit card number. Finally, both have a telephone number. The first two and the last column are common to all customers, but the other attributes are specific to each of the subtypes. Nevertheless they are aligned consistently in one subtype. Therefore we want to call these sets of cells columns as well. So in the example, name as well as payment mode are columns.

Assume the entries are supposed to be ordered alphabetically by name. Then the list contains private and corporate customers in arbitrary order. The table constructs of many presentation technologies such as HTML, LaTeX, or even GUI toolkits allow for automatic adjustment of column widths; but there is a problem with the columns specific to the subtypes. It should be possible to adjust the relative widths of one subtype independently from the other subtypes. For example, if the payment mode field of corporate customers would become smaller, it should not affect the private customers.

If the example table is realized with HTML, one can create a partial solution. In HTML, each cell element has modifiers `colspan` and `rowspan` that create non-simple cells through the union of adjacent rows or columns by giving an integer specifying the number of joined simple cells in each direction. 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. The resulting cell spans over several rows or columns. The attribute account number and the attribute card number would have colspan 2, which means that they actually merge two horizontally adjacent cells. However, this solution is, however, incomplete: it requires to fix the total order of column borders, which is an unnatural solution. Assume for example, after some minor changes in the formatting of the cell contents the account number field would need less space than the credit card field, as shown in Fig. 7. This layout cannot be produced by the HTML renderer with the same colspan declarations; the colspan attribute would have to be moved to the cells of a different row. The problem of fixing the total order of the column borders becomes even more apparent if there are many independent column widths in different subtypes. Hence, this solution is unsatisfactory with respect to dynamic adjustment of table widths. Furthermore, the solution is unsatisfactory with respect to an abstraction principle: the code for the rows representing different subtypes is dependent on the other subtypes with regard to the colspan specifications, and can therefore not be modified independently. Both disadvantages can be overcome with the partially ordered tabstops and areas of ALM. ALM generalizes colspan and rowspan by allowing areas that extend between any two x-tabs and any two y-tabs.

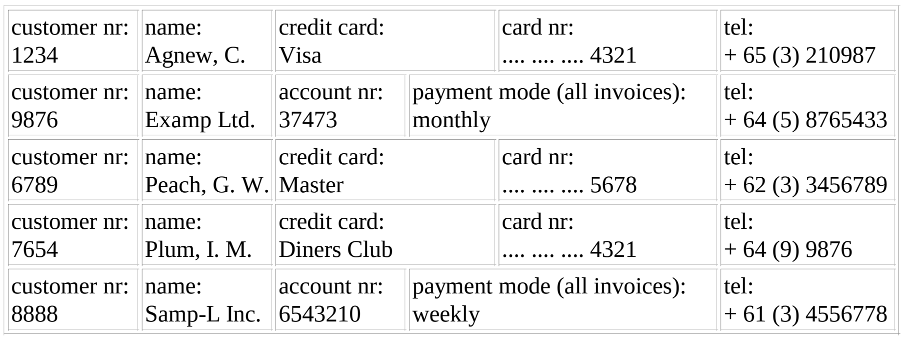
In ALM we can let the layout engine adjust the width of the columns that are specific to either private or corporate customer to the actual required size for these specific attributes. The order of tabstops can be specified partially, as is illustrated in Fig. 8 for the x-tabstops of the example. x_3 is the tabstop between the columns for credit card type and an a credit card number; x_4 is the one between the columns for account number and payment mode. It is simply not defined whether x_3 comes before x_4 or vice versa, thus leaving more flexibility for the layout engine, which can adjust the respective columns to the space that is actually needed in each of the columns. In contrast to other layout methods, a GUI developer does not have to fix the order of tabstops if it is irrelevant for the layout or unknown in advance.

With ALM the order of specification of the different areas is independent from the position of the areas in the presentation. Note, however, that the shortcomings of current tabular layouts discussed here are not caused by this circumstance alone. The crucial dependency on a total order is caused by the



customer nr: 1234	name: Agnew, C.	credit card: Visa	card nr: 4321 4321 4321 4321	tel: + 65 (3) 210987
customer nr: 9876	name: Examp Ltd.	account nr: 37473	payment mode: monthly	tel: + 64 (5) 8765433
customer nr: 6789	name: Peach, G. W.	credit card: Master	card nr: 9876 5432 1234 5678	tel: + 62 (3) 3456789
customer nr: 7654	name: Plum, I. M.	credit card: Diners Club	card nr: 2468 1357 9876 4321	tel: + 64 (9) 9876
customer nr: 8888	name: Samp-L Inc.	account nr: 6543210	payment mode: weekly	tel: + 61 (3) 4556778

Figure 6: Example of an HTML table using colspan. It shows two different types of rows.



customer nr: 1234	name: Agnew, C.	credit card: Visa	card nr: 4321	tel: + 65 (3) 210987
customer nr: 9876	name: Examp Ltd.	account nr: 37473	payment mode (all invoices): monthly	tel: + 64 (5) 8765433
customer nr: 6789	name: Peach, G. W.	credit card: Master	card nr: 5678	tel: + 62 (3) 3456789
customer nr: 7654	name: Plum, I. M.	credit card: Diners Club	card nr: 4321	tel: + 64 (9) 9876
customer nr: 8888	name: Samp-L Inc.	account nr: 6543210	payment mode (all invoices): weekly	tel: + 61 (3) 4556778

Figure 7: This layout cannot be achieved with the same colspan settings.

colspan concept. This concept makes one column dependent on other, unrelated tabstops that just happen to be in its interval of tabstops, and this creates the maintainability drawback addressed here.

5.2 The Qualitative Layout Model

The qualitative layout model defines a partial order on x-tabstops and y-tabstops, respectively. The overall tabular structure of the layout can be described just with this ordinal information. The partial order of tabstops is given by the definition of rectangular *areas*, which are bound by a pair of x-tabstops and a pair of y-tabstops each. We want to define an area a 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. The *content* can be

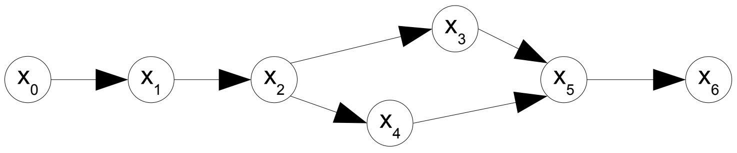


Figure 8: Partial order of x-tabstops for the example table.

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.

Areas are similar to "bricks" in the Bricklayer layout system [5], but the tabstops of their bounds are given directly, making it easier to define grid-like layouts, which are very common in graphical design. The focus is on the location of the tabstops rather than the location of each area. The areas are assigned to the tabstops.

5.3 Partial Orders on Tabstops

The qualitative layout model specifies a tabular layout by a set of areas A. This set creates a partial order of the x-tabstops and the y-tabstops, respectively. Each area names a left and right x-tabstop, x_1 and x_2, and an upper and lower y-tabstop, y_1 and y_2. Therefore it contributes an edge $x_1 \leq x_2$ to the partial order of x-tabstops, and an edge $y_1 \leq y_2$ to the partial order of y-tabstops. By topological sorting of the x- and y-tabstops according to the two partial orders, a topology of a tabular layout can be inferred. Furthermore, additional ordinal constraints in the form of equations or inequalities between two tabstops could also be added explicitly.

In most other approaches the elements in a layout are ordered totally, e.g. it is fixed if a control begins before, after or exactly on the beginning of another control. Our approach allows a GUI designer to specify the order of elements only partially, i.e. only when it is deemed necessary. A specifies just as much ordinal information as is considered important, which prevents overspecification of a GUI.

As a result, there are cases where the specification is ambiguous and different topological sortings are possible. This is intended, as it leaves more flexibility for the layout engine to optimize the layout when adjusting it to content dimensions and screen space. Even if a layout specification is ambiguous, a deterministic layout engine will always produce the same result.

A qualitative layout specification L is a list of areas and a set of additional ordinal constraints given by the user. The closure of L is the transitive closure on the ordinal constraints in L. A refinement L' of L is a specification that has at least one additional constraint that is not in the transitive closure of L. A refinement S of L where the x- and y-tabstops are totally ordered is called a qualitative solution of L.

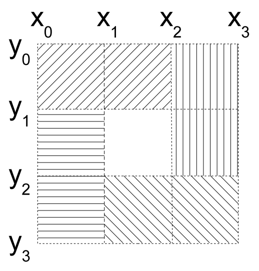


Figure 9: Equidistantly scaled layout for the ordinal data in Sect. 5.3.

If the ordinal information is conflicting, such conflicts can be detected and reported to the user. Conflicts occur if there are cycles in the ordinal data between tabstops that cannot have the same position. For example, if the ordinal data contains the edges $x_1 \leq x_2 \leq x_1$, and we know that there is an area between x_1 and x_2 that cannot have width zero, then there cannot be a valid layout.

To illustrate the qualitative layout model, let us consider the following set A of areas:

$$\begin{aligned} A \;=\; & \{(x_0, y_0, x_2, y_1, \mathit{diagonal}_1), (x_2, y_0, x_3, y_2, \mathit{vertical}), \\ & (x_1, y_2, x_3, y_3, \mathit{diagonal}_2), (x_0, y_1, x_1, y_3, \mathit{horizontal}), \\ & (x_1, y_1, x_2, y_2, \mathit{empty})\} \end{aligned}$$

This input creates a tabular layout as depicted in Fig. 9. The constants for the content of the areas, *horizontal*, *vertical*, $\mathit{diagonal}_1$ and $\mathit{diagonal}_2$, denote the type of hatching that marks the respective area in the figure. Although this is not implied in the ordinal data, the coordinates of the x- and y-tabstops in the figure were scaled equidistantly, so that the widths and heights of all the columns and rows, respectively, are the same.

5.4 Representing Partial Orders

As already mentioned, all tabstops correspond to variables that eventually contain their locations on the x- or y-axis of the GUI's coordinate system. With this in mind, it is very easy to formulate a set of linear constraints that describe the ordinal constraints given by an area. For each area $(x_1, y_1, x_2, y_2, \mathit{content})$ we introduce two constraints:

$$x_1 \leq x_2, \; y_1 \leq y_2.$$

ALM furthermore offers tabstops with given values that represent the borders of a GUI: $x_{left} = 0$ for the left border, $y_{top} = 0$ for the top border, $x_{right} = w_{total}$ for the right border, and $y_{bottom} = h_{total}$ for the bottom border. w_{total} and h_{total} are the width and height of the GUI, which are given by the environment.

5.5 Metric Properties of Areas

There are several aspects to the optimal layout of the areas of a GUI. For one thing, we would like a system that does not waste screen real-estate unnecessarily. Furthermore, we would like the system to be modular, so that each area can contain properties describing its demands while having the system as a whole find a global solution that respects the desires of each area as much as possible.

ALM supports the specification of properties 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.

Each area has penalty coefficients. These coefficients determine the difficulty with which the dimensions of the area deviate from its preferred size when the layout engine has to resize areas. By varying the penalty coefficients, the space demands of areas that are considered more important can be given preference over those of less important areas. Areas that have high penalty coefficients are only squeezed or stretched with difficulty, while areas with small penalty coefficients change their size more easily.

If screen space is scarce and the areas have to be squeezed to fit, the areas with small penalty coefficients will yield first and allow the areas with larger ones to take up the space they prefer to have. Like this, penalty coefficients can be used much like priorities, according to the degree to which an area should keep its preferred size. If the areas with low penalty coefficients cannot be squeezed any further and screen space is still tight, then the areas with the next higher penalty coefficients are squeezed as well.

ALM allows GUI developers to specify penalty coefficients on a very detailed level: not only can there be different penalty coefficients for the horizontal and the vertical axes, but also different coefficients for squeezing and for stretching an area, respectively. This way, the designer of a GUI can, for example, give important areas a high penalty for squeezing and a low penalty for stretching, so that the area will take up available space, but not yield easily when there is not enough.

If there were only one penalty coefficient for both stretching and squeezing, some resizing behavior could not be specified in a satisfactory way. An area with high penalty coefficient would not yield easily when there is little screen space available, but also would not take up available space easily. Sometimes we want areas not to yield easily to the pressure of others, but to take up as much space as they can. Or, we might have some decorative graphical elements in a GUI that should vanish as soon as screen space becomes scarce, but grow only very reluctantly in size. In both cases, we need very different penalty coefficients for squeezing and stretching.

Each area has five main parameters:

Preferred size s_{pref}**:** the preferred width w_{pref} and height h_{pref} of an area.

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.

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.

While it is possible to imagine several different sets of parameters, we believe this set to be both sufficiently general so as to obtain excellent optimal solutions, and sufficiently simple so as to be manageable to the designer of the GUI.

We would like to point out that the preferred size and penalty coefficients for areas are just absolute soft constraints of the form $w = w_{pref}$ and $h = h_{pref}$, with w being the width and h being the height of an area. The properties p_{shrink} and p_{expand} of an area directly correspond to the penalty parameters p_{pos} and p_{neg} of the respective soft constraint.

Often, it is reasonable to choose the penalty coefficients of an area dependent on the control that it contains. Therefore, ALM offers default values for each type of control. This means that when the penalty settings are left out for an area, those settings can be chosen automatically depending on the control that is contained in the area. For example, editable controls such as list boxes can in general make much better use of additional space than controls that only display a fixed amount of information, such as textual labels. Therefore, such controls have lower coefficients for expand penalty by default.

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. In addition to making the specification process easier, the default values serve as a guide when developers want to customize parameters manually, i.e. new penalty coefficients are chosen in relation to the predefined default coefficients.

5.6 Representing the Metric Properties of Areas

With the knowledge of how to represent soft constraints, representing the metric properties of areas is easy. For the minimum and maximum width and height of an area we add the following constraints:

$$w \geq w_{min},\ h \geq h_{min},\ w \leq w_{max},\ h \leq h_{max}$$

with w being the width of the area, i.e. $w = x_2 - x_1$, and h being its height, i.e. $h = y_2 - y_1$. If a minimum or maximum for a dimension is not given, then the respective constraint is simply omitted.

As described in the previous section, the preferred size of an area corresponds to absolute soft constraints of the form $w = w_{pref}$ and $h = h_{pref}$. The properties p_{shrink} and p_{expand} directly correspond to the penalty parameters p_{neg} and p_{pos} of the respective soft constraint.

Besides the aforementioned parameters, ALM offers auxiliary area parameters such as settings for the margins between the borders of an area and the borders of the content in the area, or the horizontal and vertical alignment of the content in an area. The implementation of these parameters is slightly more sophisticated as they require the insertion of an additional area within the original area's bounds and linear constraints between those two areas for margins etc. However, such additional areas are only added internally on demand, i.e. when these parameters are actually used, and do not affect the way the user perceives or uses areas. That is, the user is shielded from these complexities.

6 Rows and Columns

The organization of content in rows and columns is very common and well known from spreadsheets, therefore ALM offers abstractions for rows and columns. As described in the previous section, the location of an area is given by an x- and a y-interval in the GUI coordinate system, each specified by a pair of tabstops. Rows and columns are abstractions that allow several areas to use the same intervals. They are first-class entities that can be used to specify important properties of those intervals more easily. In the following, we discuss properties of columns. The concepts used for rows and the concepts used for columns are intrinsically symmetric to each other, therefore the discussion applies to rows as well.

Columns are characterized by the x interval that they represent. Hence two important variables of a column c are the left and right border $c.left$ and $c.right$ of the interval. An operation for adding an area a to a column c is defined that automatically creates the necessary constraints. When adding an area to a column, alignment parameters can be set: similar to a spreadsheet, an area in a column does not necessarily fill out the column width completely. ALM allows GUI developers to set parameters for horizontal alignment, such as making an area in a column left-aligned, right-aligned, centered or aligned to both sides. Corresponding constraints are added automatically. When adding areas to a row, analogous parameters for vertical alignment can be set.

Other important parameters of a column relate to the ordering of sets of columns. We allow the user to give constraints that specify an order between two columns c and d: a constraint $c < d$ expresses that the row c is completely left of row d, i.e. this is translated to $c.right < d.left$. Alternatively, a constraint of the form $c|d$ can be used that expresses that the two columns are directly adjacent, i.e. $c.right = d.left$, as shown in Fig. 10. This makes it very easy

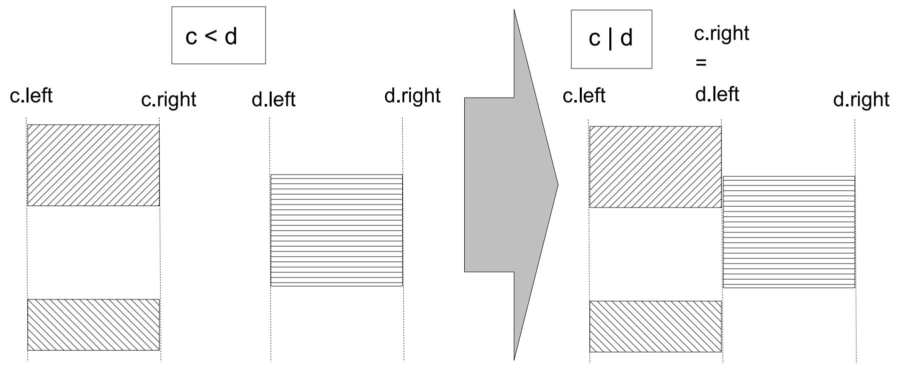


Figure 10: Columns can be compared and declared to be adjacent.

for GUI developers to change the order of columns without having to consider individual tabstops. The tabstops are taken care of by the layout manager.

A further abstraction becomes important in the context of applications that use tables such as the one shown in Fig. 6. The rows present instances of two different customer types. Every row of the same customer type uses exactly the same set of columns, but the sets of columns used for each type differ. Each type uses its own *grid*, i.e. its own set of columns and set of rows. In a grid, there must be one area in every column of every row. However, the columns and rows of a grid need not be adjacent to each other, and different grids may share several rows and columns. By specifying two grids with disjoint rows and some common columns, the layout in Fig. 6 can be described. This is much easier than using ordinary table constructs, and leaves the two grids independent of each other. The two grids can be maintained independently, commonalities between grids can be factored out and defined separately, and an order between rows and columns of different grids needs only be specified if this is necessary for the application.

7 Examples

This section describes two small examples. The first example demonstrates areas with additional linear constraints. The set of areas A for this example is taken from Sect. 5.3, and the linear constraints are the following ones:

$$C = \{y_1 = 50, -2x_1 - x_2 + x_3 = 0\}.$$

In order to illustrate the dynamic adaptation of the layout, Fig. 11 shows several screenshots of the GUI, each rendered in a window of different size. No matter how we resize the window that contains the UI, the top-left area will always end at y-position 50, and the top-right area will always be twice as wide as the bottom-left one, as specified in the linear constraints.

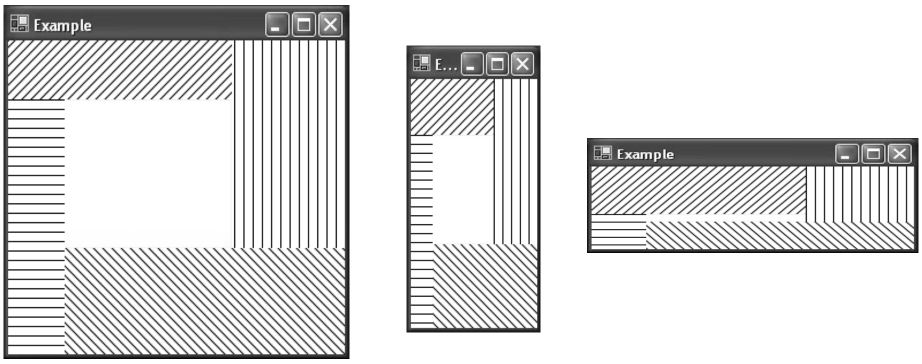


Figure 11: Example layout with linear constraints, adjusted to window resizing.

The second example demonstrates the effect of penalty constants in layouts. The layout is defined by the set of areas $A = \{a_1, a_2, a_3, a_4\}$ with

$$\begin{aligned} a_1 &= (x_0, y_0, x_1, y_1, button_1), \\ a_2 &= (x_1, y_0, x_2, y_1, button_2), \\ a_3 &= (x_0, y_1, x_2, y_2, button_3), \\ a_4 &= (x_0, y_2, x_2, y_3, textBox) \end{aligned}$$

and a single linear constraint

$$x_1 - x_0 = x_2 - x_1.$$

For a_1 and a_2 $p_{expand} = 0.6$, for a_3 $p_{expand} = 0.5$, and for a_4 $p_{expand} = 0.4$. Furthermore, a_4 has a maximum height of 50.

Figure 12 shows three screenshots of this GUI. We see that, as specified in the linear constraint, the two buttons at the top of the GUI always have the same width. When we increase the total height of the GUI a bit, the additional space is first given to the textbox at the bottom of the GUI. This is because the area that contains the textbox, a_4, has the smallest penalty for expanding the size over its preferred size. However, when we keep increasing the height of the window, the maximum height of a_4 is reached and a_3, the area with the second lowest p_{expand}, is given the additional space.

8 Patterns in the ALM

Since a focus of this work is to identify powerful abstractions for constraint modeling, we want to discuss ways in which such abstractions are found and motivated in practice. One way how abstractions can be found naturally is to start with examples of good models. From this one can obtain guidelines which improve the quality of a model definition. One way how such guidelines are captured in Software Engineering, is to present them as patterns. Patterns are

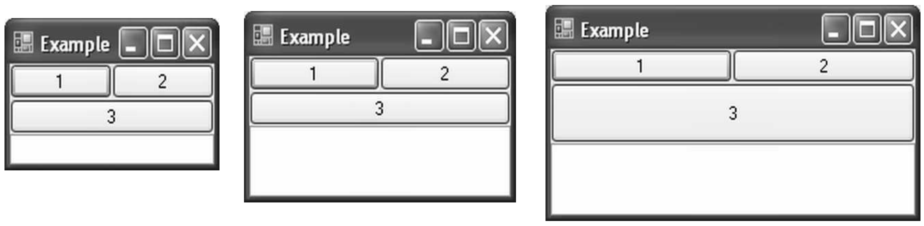


Figure 12: Example layout with linear constraints and penalty constants, adjusted to window resizing.

considered a particular high level form of abstraction. A pattern is a recurring solution to a common type of problem. It generally encodes good practice. The application of patterns typically does not require automatic tool support. In this section we want to understand how some aspects of good modeling would look like if they are only realized in the form of patterns in a model. This will then shed light on the usefulness of the other abstractions presented in this article, because then many abstractions described in this work can be understood as automated realizations of such patterns.

8.1 An Overview of Patterns in Tabular Layout

The first pattern used in the Auckland Layout Manager is the use of variables as tabstops. With our fundamental concept of tabstops, it is possible to define some of the remaining features of other layouts as mere patterns of tabstop use. Some widespread parameters of table layout can be reduced to the concept of tabstops. Also the use of constants and rank units could well be seen as a use of patterns, but because of their close relation to soft constraints we have discussed them in a different context.

A useful pattern that enhances maintainability of table definitions is the *interval tabstop* pattern. It enables easy rearrangement of columns and rows. Let us consider this pattern for columns: for each column, two tabstops are defined, the left and the right. All cells of this column use these two tabstops as their left and right x-tabstops. The order of the columns within the table is then fixed *afterwards*, by adding constraints that identify the start tabstop of a column with the end tabstop of the previous column. In this manner, the order in which the columns appear in the table is encapsulated in a separate set of constraints, independent of the definition of the columns themselves. Consequently, the order of the columns can be changed very easily without interfering with the rest of the specification. In the same manner, we can specify the order of the rows of a table in a separate set of constraints. This pattern is actually the foundation of our abstractions for rows and columns described in Sect. 6.

Further patterns could be identified for table embellishment: for the padding of a cell, two tabstops can be introduced at each side of the cell, and the absolute width can be fixed. Other concepts, such as the global cell spacing of the table, can then be seen as being paired with a generative concept. For expressing global cell spacing we have to introduce new tabstops at all previously defined tabstops and to define auxiliary cells which implement the padding. All these cells are set to the same width, and this width is the cell spacing. A natural way to deal with this relationship would be to offer the concept of global cell spacing as an abstraction, that is, a separate notion just like in HTML tables that is convenient for the user. In the rendering engine one could then use the translation into the more fundamental representation based solely on tabstops.

Another example for a common parameter in table constructs is the existence and appearance of delimiters and borders between and around parts of the table. Like padding, any kind of decorations, be it simple delimiting lines or elaborate frames, can be supported with additional tabstops and areas adjacent to the existing ones. For example, a line can be represented as a thin unicolored area. Again, it is conceivable to encode common table styles in higher-level constructs that generate the lower-level representation of tabstops and areas.

8.2 Advantages and Drawbacks of Patterns

The pattern approach does not immediately yield a high degree of automation, therefore we did not restrain ourselves to the use patterns but provided some of these abstractions in the implementation of the Auckland Layout Manager as API definitions that can be used in specifying a layout. It has to be said though that patterns have advantages as well: one advantage is the semantic austerity of the approach, which leads to a great clarity of the definitions. Take the interval tabstop pattern above as an example. It provides already an abstraction for rows and columns. It does not allow us to define new operators as did the full abstractions for rows and columns that we have presented earlier, but in some sense the user will have to read the definition of these operators anyway in order to understand their semantics. Patterns seem to be particularly powerful in the area of constraint programming, because there is one fundamental pattern in constraint programming that is extremely powerful in itself, and we want to call this the *separate variable* pattern. This pattern can be defined as follows: wherever you can identify several clearly distinguishable uses of the same variable, there you should introduce one separate variable for each use and add explicit equality constraints that set these variables equal. This is the generalization of the interval pattern discussed above. For many abstractions the main insight lies in the recognition of such separate uses of a variable. For many abstractions it is very interesting to consider the actual model that will be generated and to realize the patterns that are present in this model due to the use of higher level API functionality.

9 Related Work

In order to create document layouts, most people use simple desktop publishing software such as, for example, MS Word or MS Powerpoint. These programs offer a medium range of features for arranging graphical elements and thus creating a layout suitable for a particular kind of media. The layout paradigm commonly used in such software is a very simple one: the user can arrange graphical elements on absolute coordinates of a viewport and modify their appearance by changing a predefined set of properties for each of them. It is not possible to formulate constraints, but merely possible to change coordinates in an operational manner such as, for example, aligning an element. This means that the layout usually does not adapt well to any changes of the context, such as changes of the overall size.

Tools with functionality for GUI design, e.g., MS Visual Studio, usually take the same approach and therefore suffer from the same shortcomings. But since re-layout of GUIs in resizable windows is a common problem, there exist some common solutions, which are made explicit in the Java framework in the form of layout manager classes. Besides some rather simple layout managers, one of the most powerful ones is the GridBag layout manager, which allows arranging GUI widgets in a tabular manner. During runtime, the layout manager is an object that is updated in order to populate the cells. The simple cells are numbered, and each actual compound cell is inserted with a command specifying one corner and the size of the cell. Each cell can specify a set of parameters that are called constraints, for padding and dimensioning of the cell. However, programming of layouts with the GridBag and similar constructs is not easy, requires programming skills and also good knowledge of the particular UI technology used.

Tables in text documents and the widespread GridBag layout [29] represent identical layout concepts. The HTML table construct [26] is a good representative. The colspan and rowspan attributes have been explained in Sect. 5. Furthermore, it features a range of additional table parameters such as horizontal and vertical cell spacing, outset of the table, inset of cells, and parameters for single cells such as padding and cell dimensions. HTML tables offer the possibility to specify cell dimensions either absolute in pixels or relative by giving percentages. Similar tables can be found in the earlier LaTeX tabular environment, where the table content is given with a markup language as well. Non-simple cells are created by special LaTeX commands such as `multicolumn`, and the LaTeX framework defines a set of default values, e.g., for cell padding. Several frameworks support definition of sizes that are tolerant and can change according to the rest of the layout, and the usual metaphor is that of a compressible placeholder. Examples are the variable measures in LaTeX and the Java SpringLayout. The concept of tolerances is again an abstraction, similar to padding etc., and can be defined in our constraint methodology by using auxiliary tabstops. In all the mentioned table concepts the cells have to be specified in a strict order, i.e., the tabular layouts use totally ordered columns and rows. In order to give tables more flexibility, they feature cells that are not simple. As discussed in Sect. 5, our table concept does not require non-simple cells, but offers superior flexibility by allowing only a partial order of tabstops.

The scientific community has examined the concepts of layout already very early. In [30] a graphics typesetting language is described that employs a number of different constraints, also nonlinear ones, in order to create images. Although very powerful, solving nonlinear constraints can also be very slow and is not necessary for a good layout. Such languages were made for creating complex images rather than for layout and, in our opinion, they do not offer the level of abstraction and ease of use that would be desirable for our task. Other approaches that do focus on layout, such as the one described in [10], support only basic layout schemes such as simple rows and columns or a flow layout, which makes them insufficient for elaborate UI design. Then, there are approaches that focus on the theory of automatic optimization of tabular layouts rather than their specification. In [1], for example, the geometric problem of space-efficient table layout is examined from a theoretical point of view, but the model used for the specification of the tables is rather simplistic.

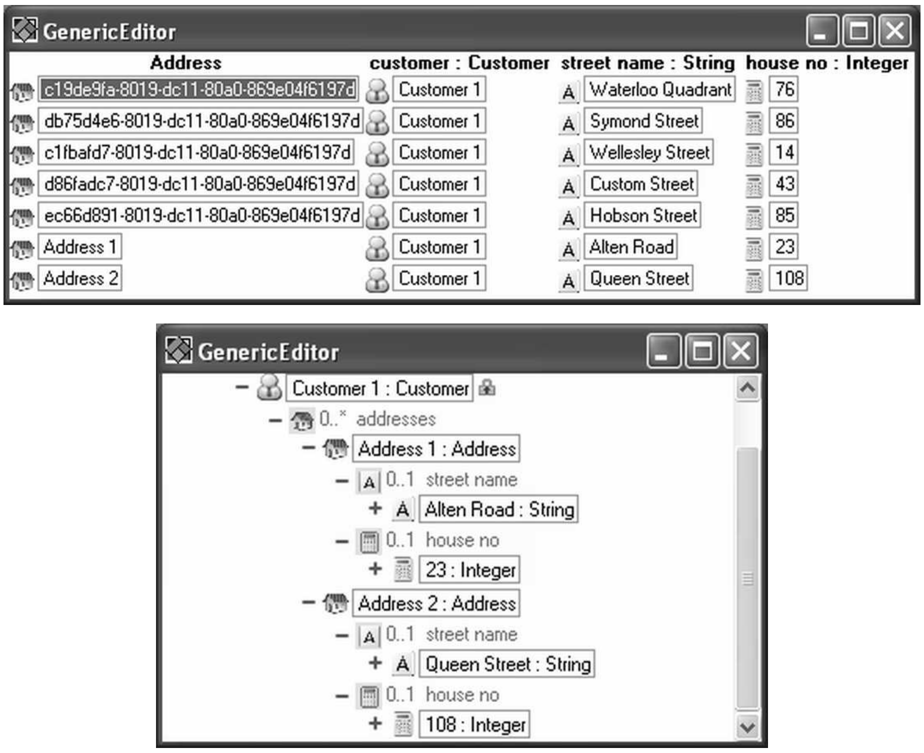
The idea to use linear constraints for UI layout is not new and has been described, for example, in [9, 2]. But in contrast to our idea, these approaches often make only use of such constraints in order to describe a layout. While it would be possible to map our approach onto one that uses linear inequalities instead of partial orders, we find it much more natural to specify the ordinal information implicitly by just defining areas. Specifying numerical constraints can be cumbersome if the topology of the layout is not yet established, and a user should only be required to work out numerical equations where it is really necessary.

Also the idea to specify the topological and metric properties of the layout has been described before. For example, [3] proposes a grid for the definition of an "abstract layout" and a constraint solver to calculate the concrete layout. But in contrast to our approach, a grid describes the topology of a layout as a total order, which is sometimes not flexible enough. Nevertheless, the grid layout is widespread and, for example, also used in the most popular professional publishing program, QuarkExpress, and in [19, 18]. The latter describes the automatic adaptation of documents to different document formats.

Some approaches use quite sophisticated models to represent the layout of tables. For example, the model in [28] uses an object network of about 7 connected layers to describe the physical layout. Our approach tries try to keep the complexity involved in layout specification to a minimum. The same paper distinguishes, like many others, between the logical, content-related and the physical, appearance-related information contained in a table. In this paper we are merely concerned with the physical side but it is conceivable to put further abstractions on top of our model for the generation of layouts for collections of content.

For constraint-based drawing tools, dedicated constraint solving approaches have been developed, particularly making use of incremental change [13]. Such approaches are able to handle more general constraints than linear constraints. Since the Auckland Layout Manager serves however as a specification and not only as a single implemented product, we consider it advantageous to stick with linear constraints, since they are certainly powerful enough to model the constraints desired in this application.

Our work on ALM is based on previous work that was published in [22]. Our


GenericEditor

Address	customer : Customer	street name : String	house no : Integer
c19de9fa-8019-dc11-80a0-869e04f6197d	Customer 1	Waterloo Quadrant	76
db75d4e6-8019-dc11-80a0-869e04f6197d	Customer 1	Symond Street	86
c1fbafd7-8019-dc11-80a0-869e04f6197d	Customer 1	Wellesley Street	14
d86fadc7-8019-dc11-80a0-869e04f6197d	Customer 1	Custom Street	43
ec66d891-8019-dc11-80a0-869e04f6197d	Customer 1	Hobson Street	85
Address 1	Customer 1	Alten Road	23
Address 2	Customer 1	Queen Street	108

GenericEditor

```
− Customer 1 : Customer
  − 0..* addresses
    − Address 1 : Address
      − 0..1 street name
        + Alten Road : String
      − 0..1 house no
        + 23 : Integer
    − Address 2 : Address
      − 0..1 street name
        + Queen Street : String
      − 0..1 house no
        + 108 : Integer
```

Figure 13: Screenshot of a structured data editor using ALM.

old layout engine is not based on linear programming, but uses a combination of topological graph sorting and Gauss-Seidel linear solving instead. This is a lot less powerful than ALM, e.g. it does not support soft constraints or penalty functions, and formally not as coherent as the linear programming approach. These essential shortcomings became clear to us when using the layout engine in practice, and led to the development of ALM.

10 Evaluation

We have used ALM to implement a multi-view structured data editing system called AP1 [20], which is shown in the screenshots of Fig. 13. AP1 is based on a structured data repository that can be accessed by multiple users simultaneously. The repository uses the parsimonious data model (PDM) [12], which structures data using entity types and relation types. Entity types are sets of elements called instances. Relation types are sets of links that connect the instances of two entity types. Each end of a relation type is identified by a role. Two dynamic, editable views were implemented using ALM: a table view, which is shown at the top of Fig. 13, and a tree view, which is shown at the bottom.

The screenshot of the table view shows instances of type Address, one in each row. The first column "Address" contains the identifier of each Address instance. The other columns show instances that are linked to the Address instances using a particular role. For example, each Address instance has a link to an instance of type Customer. Hence, there is a column "customer:Customer" listing Customer instances that are connected to the respective Address instances through a role "customer". Analogously, we have columns that represent String and Integer instances for street names and house numbers that are linked to each Address. The tree view was implemented using the abstractions for rows and columns. ALM offers operations for insertion and removal of rows and columns, therefore updates of the view were easy to implement. The ALM features for alignment of controls in areas were used to place the column headers in the center of their column, and align instances at the left column borders. As described in Sect. 6, ALM supports reordering of rows and columns. This made it relatively straightforward to enable sorting of rows and reordering of columns in the view.

The screenshot of the tree view shows a Customer instance "Customer 1" and the Address instances that are connected to it through role "addresses". Each address has a role "street name" that is used to connect a String instance, and a role "house no" that is used to connect an Integer instance. Instances and roles are arranged in rows. The indentation of child nodes is expressed as linear constraints. The plus and minus signs on the left side of roles and instances can be used for elision, i.e. in order to hide or expose subtrees. This was implemented using the operations for removal and insertion of rows.

When first implementing the editing system, we were trying to use the standard C# controls for tables and trees. However, they put strong restrictions on the way data elements can be represented. For example, a tree node can basically just contain a single icon and a textual label, and has to have a fixed height. As a result, it was not possible to represent instances in a flexible manner. This was one of the reasons why we decided to implement the views with ALM. Another reason is the inability of the standard controls to adjust themselves to the size of their content. For example, the widths of columns have to be set explicitly and are not adjusted according to the width of the items they contain. This results in columns that take up more screen space than necessary, thus wasting screen real estate, and columns that make their content unreadable by truncating it. Furthermore, the standard controls do not properly support view configuration tasks such as reordering and sorting.

When switching to ALM, we first implemented the views using merely tabs, areas and linear constraints. It became clear very soon that the level of abstraction of these concepts was too low in order to achieve a clean, maintainable implementation. Using the constructs for rows and columns improved the level of abstraction drastically, and made the implementation much easier. Instead of having to worry about individual tabstops and areas, it was now possible to manage them grouped in columns and rows. Most operations such as placement, removal and insertion could be done on that level. In terms of lines of code, the parts of the implementation concerned with view layout shrunk to less than half their original size. Testing and debugging of the layout code took about a day, compared to about a week of debugging when using the original approach.

While mostly manipulating layouts on the level of rows and columns, it was still necessary to access lower levels such as tabstops and constraints. For example, this was necessary for implementing non-tabular layout characteristics such as the indentation in the tree view. Because the semantics of the row and column constructs were well-defined in terms of linear programming, the combination of specifications from different levels of abstraction turned out to be unproblematic. That is, adding constraints manually to the specification did not cause any surprises such as unexpected interactions.

A significant advantage of ALM became apparent while implementing the tree view: using a constraint-based approach such as ALM, it is much easier to achieve modularity in the layout specification [23]. Before using ALM, we implemented the tree view using recursively nested panels. Each instance and role was represented in its own panel, which also contained the panels of its children. That is, each subtree was represented as a panel, and the tree structure was reflected in the containment hierarchy, which could be very deep. Each panel control took care of the representation of a single tree node and its direct children. Apart from being rather slow, this approach worked only as long as there were no dependencies between different branches of the containment hierarchy. For example, if subtrees at different places should share the same horizontal alignment, then this was extremely hard to express. It meant that the panels – although not directly related – had to communicate to make sure they used the same alignment. In other words, the concept of a hierarchical partitioning of a GUI cannot always result in a modular design in which the parts are independent of each other. However, ALM is not bound to such a hierarchical structure: the specification modules are arbitrary sets of constraints that may relate to any tabstop in the GUI. The composition of the modules is simply the union of those sets. If we want to express a crosscutting concern such as a uniform horizontal alignment across different subtrees in the tree view, then we can do so by specifying a new modular set of constraints and inserting it into the set of all constraints.

Multiple users can simultaneously work on the same data in the repository, each using their own client with their own views. In this situation, notifications about changes in the data are synchronously transmitted to every client as soon as they are committed to the repository. Upon receiving such a change notification, each client immediately updates all its views. As a result, both the tree and the table view are highly dynamic. With many users working on the same data, many changes occur and views keep changing all the time. However, individual data changes and the corresponding changes in the layout of a view are usually quite small. Through incremental constraint solving layout calculation times could be kept low, so that it was possible to deal with such frequent changes of the underlying layout specification without affecting the user interface negatively. As it turned out, the bottleneck of the presentation layer was not the layout engine but the .net graphics rendering system.

Despite the good performance of our ALM implementation, it is unclear if ALM would be a preferable option for devices with low resources, e.g. portable devices such as PDAs or mobile phones. ALM is still slower than most of the simple layout managers commonly used in applications, and it is built on top of a linear solver, which increases the memory footprint of an application (in our implementation by about 300KB). Furthermore, ALM is not only more powerful but also more complex than many other simple layout techniques, which means that developers have to invest time to get used to it.

Nowadays there is a fairly good range of tools supporting the construction of GUIs, often in a WYSIWIG manner. Although support for layout managers in such tools is often insufficient, they do nevertheless contribute significantly to the efficiency of the GUI development process. As a fairly new layout model, tool support for ALM has not fully matured yet, but is in the prototype stage. As a more sophisticated layout model, such tool support is not as straightforward as it is for simple layout managers. However, there is a reverse engineering technique for ALM that makes it possible to use existing GUI construction tools and automatically recover an ALM specification afterwards [21].

Sometimes layout specifications contain errors such as conflicting constraints, or simply constraints that cause a malformed or undesired layout. In order to make debugging of specifications easier, ALM makes textual representations of all the layout constructs at all the different abstraction levels readily available. Like this, developers can examine a specification or a subset of it in human-readable form. It is also possible to export the underlying linear programming specification in standardized formats, so that the specification can be used from other mathematical tools such as MATLAB. This makes it possible to use external tools for analyzing specifications and detecting faults. As future work, it would be desirable to integrate support for debugging into a visual layout specification tool.

To find out how efficient it is to specify GUI layout using ALM, compared to other layout models, an experimental study is necessary. Such a study could measure indicators such as the time and accuracy with which test subjects complete certain layout specification tasks, using different layout approaches. Such studies are hard to perform and have to be designed very well in order to avoid potential pitfalls. The test subjects have to be chosen from a well-defined population and the sample needs to have a considerable size in order to get statistically meaningful results. Factors such as differences between the test subjects in experience and skills, and the order in which tasks are presented to test subjects need to be considered and controlled carefully. Having different groups of test subjects may improve the validity of the result, but means that more test subjects are required. The tasks need to be chosen so that they are meaningful and reflect real-world requirements.

11 Conclusion

Constraint programming approaches to GUI layout are interesting in that they make the constraint solving process truly ubiquitous: many changes in the GUI, such as resizing of a window or changes in the presented content, lead to a reevaluation of the underlying constraints. Our vision is to bring this approach to the working developer and to supersede the weaker layout managers without constraint programming currently in use. We described what kind of abstractions are necessary to make GUI layout with linear programming easier for GUI developers, by discussing the abstractions provided by the ALM. The following abstractions are provided by ALM and were found very useful for GUI specification and GUI maintenance:

- Soft constraints with support for constraint hierarchies and approximation of arbitrary penalty functions, enabling affinity towards several particular values.
- Area specifications that impose a partial order on a grid of tabstops and offer straightforward parameters for the resizing behavior of controls.
- Abstractions for rows, columns and grids, allowing GUI developers to arrange controls similarly to widely-used spreadsheets, but with more flexibility and modularity.
- Patterns and minor abstractions for improving maintenance such as pervasive use of constant parameters, support for different units...

GUI layout systems that use linear programming without offering higher-level constructs such as the ones described here may be sufficient from a formal point of view, but fail to serve the needs of the practitioners of that domain. If constraint solving approaches are to be successful in practice, they need to offer abstractions that capture the concepts of the application domain rather than just providing a mechanism that is merely formally complete.

References

[1] Richard J. Anderson and Sumeet Sobti. The table layout problem. In *SCG '99: Proceedings of the 15th Annual Symposium on Computational Geometry*, pages 115–123. ACM Press, 1999.

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

[3] R. Beach. Tabular typography. In *Text Processing and Document Manipulation*, pages 18–33. The British Computer Society, Cambridge University Press, 1986.

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

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

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

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

[8] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp Symb. Comput.*, 5(3):223–270, 1992.

[9] Alan Borning, Kim Marriott, Peter Stuckey, and Yi 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.

[10] Joëlle Coutaz. A layout abstraction for user-system interface. *SIGCHI Bull.*, 16(3):18–24, 1985.

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

[12] Dirk Draheim and Gerald Weber. *Form-Oriented Analysis - A New Methodology to Model Form-Based Applications.* Springer, October 2004.

[13] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Commun. ACM*, 33(1):54–63, 1990.

[14] Alan M. Frisch, Brahim Hnich, Ian Miguel, Barbara M. Smith, and Toby Walsh. Towards CSP Model Reformulation at Multiple Levels of Abstraction. International Workshop on Reformulating Constraint Satisfaction Problems at CP-02, Ithaca, NY, USA, 2002.

[15] 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.

[16] H. Hosobe and S. Matsuoka. A Foundation of Solution Methods for Constraint Hierarchies. *Constraints*, 8(1):41–59, 2003.

[17] Hiroshi 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.

[18] Charles Jacobs, Wil Li, Evan Schrier, David Bargeron, and David Salesin. Adaptive document layout. *Commun. ACM*, 47(8):60–66, 2004.

[19] Charles Jacobs, Wilmot Li, Evan Schrier, David Bargeron, and David Salesin. Adaptive grid-based document layout. *ACM Trans. Graph.*, 22(3):838–847, 2003.

[20] Christof Lutteroth. AP1: A platform for model-based software engineering. In *TEAA '06: Proceedings of the 2nd International Conference on Trends in Enterprise Application Architecture*. Springer, 2006.

[21] Christof 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.

[22] Christof Lutteroth and Gerald 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.

[23] Christof Lutteroth and Gerald Weber. Modular specification of GUI layout using constraints. In *ASWEC 2008: Proceedings of the 19th Australian Software Engineering Conference*. IEEE Press, to appear.

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

[25] P. Meseguer, N. Bouhmala, T. Bouzoubaa, M. Irgens, and M. Sánchez. Current Approaches for Solving Over-Constrained Problems. *Constraints*, 8(1):9–39, 2003.

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

[27] Michael 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.

[28] Horst Silberhorn. TabulaMagica: an integrated approach to manage complex tables. In *DocEng '01: Proceedings of the 2001 ACM Symposium on Document Engineering*, pages 68–75. ACM Press, 2001.

[29] Kathy Walrath, Mary Campione, Alison Huml, and Sharon Zakhour. How to use GridBagLayout. In *The JFC Swing Tutorial: A Guide to Constructing GUIs*. Addison-Wesley Professional, February 2004.

[30] Christopher J. Van Wyk. A graphics typesetting language. In *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, pages 99–107. ACM Press, 1981.