User Interface Layout with Ordinal and Linear Constraints

Christof Lutteroth **Gerald Weber**

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

Abstract

User interfaces as well as documents use tabular layout mechanisms. The HTML table construct and the GridBag layout in Java are typical examples. There are, however, shortcomings of these mechanisms that become obvious with more advanced content like semi-structured data or object-oriented models. We present a generalized table construct that solves these shortcomings and generalizes tabular layouts to a foundation of 2D layout. We describe an algorithm for specifying and rendering user interfaces – and 2D documents in general – using simple but expressive mathematical properties. In particular, the new tabular layout is described by ordinal and linear constraints. The ordinal information makes it possible to describe the general structure of a table and merge multiple cells into rectangular areas. The linear constraints allow it to map particular points of the table to particular coordinates or specify the size of areas in an absolute or relative manner. The resulting layout engine is easy to use and can render 2D information in real-time.

Keywords: user interface design, 2D layout, formal constraints, tables

1 Introduction

Tabular layout mechanisms are central to the implementation of today's user interfaces and for the design of complex documents. The more flexible mechanisms like the HTML table construct and the Java GridBag layout are based on an underlying basic table that can be modified by merging cells. These concepts are sufficient for many purposes, but for complex data characterized, for example, by subtypes certain shortcomings become obvious.

In this paper we present a novel tabular layout with improved expressive power. We can show that this new layout avoids some shortcomings of current layout mechanisms. Our concept is based on simple mathematical notions, which capture the essential properties of a graphical, two-dimensional layout necessary to describe a GUI. Furthermore, it is powerful enough to let the UI designer specify invariants on the UI, which make it possible to render a UI in a dynamically changing environment. The layout engine underlying our approach is capable of adjusting a UI to different circumstances while retaining the invariants. The layout presented here works for graphical user interfaces as well as documents in general. It is designed with the goal of making it possible to transfer the data of other design tools to our model quite easily; the aim is that the designer can rather focus on the creative process itself than on the specific properties of particular technological implementations. In the approach presented here the use of a tabular layout as the central layout paradigm is deliberate, so there is a shift in attitude from HTML, where it was a frequent opinion that the extensive use of tables was a workaround.

In Sect. 2 we discuss the current tabular layout mechanisms and show their shortcomings. In Sect. 3 we explain how ordinal data defines the basic structure of an UI. Section 4 discusses the application of linear constraints for specifying quantitatively exact positions and proportions. Section 5 describes composition and decomposition of layouts. In Sect. 6 we discuss related work and point out some future work. The paper concludes with Sect. 7.

2 Basic Properties of Tabular Layout

In order to understand the concepts required for the layout of graphical user interfaces, one has to study their basic properties. Probably the most important one is the fact that 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 elements in their 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, like, for example, lines and paragraphs of text, and adds to a clear overall appearance.

When we consider these two properties together and imagine every edge of an element's area to be either a vertical or horizontal unbounded line, we end up with a grid in which all the other elements 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.

2.1 Shortcomings of Current Tabular Layout Mechanisms

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, like in Figure 1. 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. We call a set of adjacent columns a *column domain*; there are two column domains at the beginning and the end of the table where both types are structured identically, and one in the middle where the columns used for each type differ. We call columns that are consistently used in at least one subtype a *proper column*. In the example, name as well as payment mode are proper 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 like 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. Assume the width of the column domain with the subtype-specific attributes is fixed. Then we would expect the widths for each of the subtype-specific attributes to be adjusted in the following way: for one subtype the widths should be adjusted completely independent from the other subtypes. If, e.g., 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 creates 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. The attribute account number and the attribute card number would have colspan 2, which means that they actually merge two horizontally adjacent cells. This solution is, however, incomplete: it requires to fix the total order of column borders, which is 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, then the renderer for the table could not make use of this; the layout shown in Figure 2 cannot be produced with the same colspan declarations. 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; a further discussion can be found in Section 6.1. Hence, this solution is unsatisfactory with respect to dynamic adjustment of table widths. Furthermore, the solution with the GridBag layout 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 table construct proposed in this paper. As an added benefit in our layout, the order of specification of the different cells is fully independent from the position of the cells in the presentation. Note, however, that the shortcomings of current tabular layouts discussed here are not caused by this circumstance alone. The crucial dependency of HTML Tables as our running example on a total order in our context here is the colspan concept alone; this concept makes one proper column dependent on other, unrelated tabstops that just happen to be in its interval of tabstops, and this alone creates the maintainability drawback addressed here.

2.2 Generalized Tabular Approach

We propose a generalized table construct. The focus switches to the borders between the table elements, which are conceived as vertical and horizontal tabulators and called *x-tabstops* and *y-tabstops*. Together they are simply called tabstops, or *tabs* for short. The current rowspan technique can be transformed into one which names the tabstops delimiting a cell. The tabstops have a symbolic name. The widely used GridBag principle has the aforementioned drawback that its tabulators are totally ordered. In the new layout the tabstops are only partially ordered by their use in cells.

A table in the new layout is a list of cell specifications. These cell specifications together create the partial orders of the stops. Each cell specification names a left and right x-tabstop and an upper and lower y-tabstop. Each cell definition contributes one edge to the partial orders of x-tabstops and y-tabstops, respectively. If the relative position of two tabstops is not fixed in this way – like in Fig. 3 – their relative position may change, depending on other requirements concerning the layout.

Consider, for example, the table in Fig. 1. We have already discussed this example and know that the width of the columns that are specific to either private or corporate customer should be adjusted to the actual required size for these specific attributes. However, in HTML we had to fix the total order of tabstops, limiting the way these widths can be adjusted. Using our table approach we are able to specify the order of tabstops just partially, as is illustrated in Fig. 3 for the x-tabstops of the example. This way it is simply not defined whether x_3 comes before x_4 or vice versa, thus leaving more flexibility for layout. The user does not have to fix the order of tabstops if it is irrelevant.

These specifications have to be set in relation to the quantitative specifications of the positions of tabstops, for example the absolute position of a tabstop. The tabstop concept generalizes many aspects of conventional page and document layout, especially the concept of tabulators in paragraphs. The new layout allows for flexible vertical tabulators in paragraphs, a concept that is usually not available.

2.3 Patterns for Tabular Layout

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, like cell padding, can be reduced to the concept of tabstops. 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, like the global cell spacing of the table, can then be seen as representing a generative concept. For expressing a global cell spacing we have to introduce new tabstops at all previously defined tabstops and to define fixed width auxiliary cells which implement the padding. A natural way to deal with this relationship would be to offer the concept of a 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 delim-

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 1: Example of an HTML table using colspan

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

iters 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. A line, for example, can be represented as a thin unicoloured area. Again, it is conceivable to encode common table styles in higher-level constructs that generate the lower-level representation of tabstops and areas.

In our work we have identified a couple of useful patterns that enhance maintainability of table definition. One pattern is for example the *separate tabstop* pattern. It fosters easy rearrangement of columns and rows respectively. In this pattern, each proper column uses its own start and end x-tabstop. All cells of this column use these two symbolic tabstops as their left and right x-tabstop. The mutual position of the proper columns as a partial order is then fixed afterwards by adding constraints that identify certain start tabstops with certain end tabstops, as discussed later. The same works for rows of course.

3 Using Ordinal Information

When talking about ordinal information, it is necessary to specify on what objects and how the order is defined. In our layout algorithm, we define 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 respectively. So one can think of an area as a group of adjacent cells in the table that are merged into a rectangle to house one of the graphical elements of the UI. We want to define an area a as follows:

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

The x-tabstops x_1 and x_2 delimit the area on the x-axis; the y-tabstops y_1 and y_2 delimit it on the y-axis. The *layer* is a z-direction specification that defines the order in which the *content* of all the areas in a UI are drawn, thereby allowing multiple layers of overlapping areas, like those in Fig. 4. The layer specification can also be given as an ordinal abstract data type, but for this discussion we assume the layer to be an integer. The inclusion of overlap directly into the tabular layout represents yet another generalization of current tabular layouts in this approach.

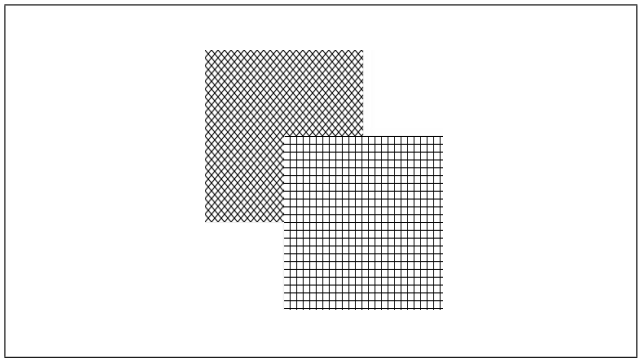

Figure 4: Layout with overlapping areas.

The basic tabular layout is defined by the set of areas A of the UI, with each area either containing one of the UI's graphical elements or, for padding, nothing. An actual tabular layout consistent with the ordinal information can already be inferred by topological sorting of the x- and y-tabstops, respectively. For topological sorting we use the zero indegree sorting algorithm, which is described, for example, in (Gross & Yellen 2003). In this algorithm we represent the partial order on the tabstops as a directed acyclic graph (DAG) and keep track of the

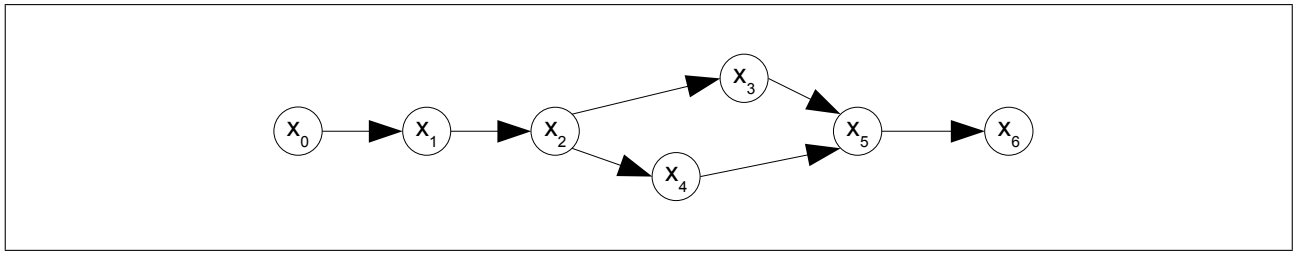

Figure 3: Partial order of x-tabstops for the example table.

in-degree, i.e., the number of inbound edges, at every vertex. The algorithm, in a simplified form, is shown in Algorithm 1.

Algorithm 1 Zero in-degree topological sorting.

```
1: sorting ← emptylist
2: while Graph G contains vertex do
3:     next ← vertex of G without predecessor
4:     sorting ← sorting + next
5:     Remove next from G
6: end while
```

Note that the set of areas A specifies just as much ordinal information as is considered important for the UI. In many cases, this will be ambiguous and may theoretically result in different topological sortings and consequently different layouts. This is intended, as it leaves flexibility for the layout engine to optimise the layout. Using the same parameters will always produce the same result, as the algorithm works deterministically. If the ordinal information is conflicting, i.e., contains a cycle like, for example, $x_1 < x_2 < x_1$, then the layout engine will detect this and report an error. In order to visualize the result obtained from just the topological sorting, we scale the x- and y-tabstops equidistantly. I.e., the width of all the rows and columns, respectively, will be the same.

To illustrate this first part of the layout algorithm, let us consider the following set A of areas:

$$\begin{aligned} A \;=\; & \{(x_0, y_0, x_2, y_1, 0, \mathit{red}), (x_2, y_0, x_3, y_2, 0, \mathit{green}), \\ & (x_1, y_2, x_3, y_3, 0, \mathit{blue}), (x_0, y_1, x_1, y_3, 0, \mathit{grey}), \\ & (x_1, y_1, x_2, y_2, 0, \mathit{empty})\} \end{aligned}$$

This input creates a tabular layout as shown in Fig. 5. Since there are no overlapping areas, all areas are located on layer 0. The constants *red*, *green*, *blue*, *grey* and *empty* indicate that each of the areas is just uniformly filled with a colour or empty. To make the illustrations clearer for print we substituted the colours by backward diagonal, vertical, forward diagonal and horizontal hatching, respectively.

4 Using Linear Constraints

Ordinal information allows us to specify the overall structure of a layout, but of course, this rather qualitative information is not enough. We also need a way to specify quantitative constraints that allow us to place tabstops exactly. For this purpose, we apply linear constraints, which capture all the properties that seem necessary for UI layout.

A layout L is therefore defined as a tuple (A, C), with A being the set of areas in the layout and C being the set of linear constraints defined on their tabstops. A tabstop in a constraint is interpreted as a variable describing its x- or y-position in the layout, respectively. So if we consider a layout with x-tabstops

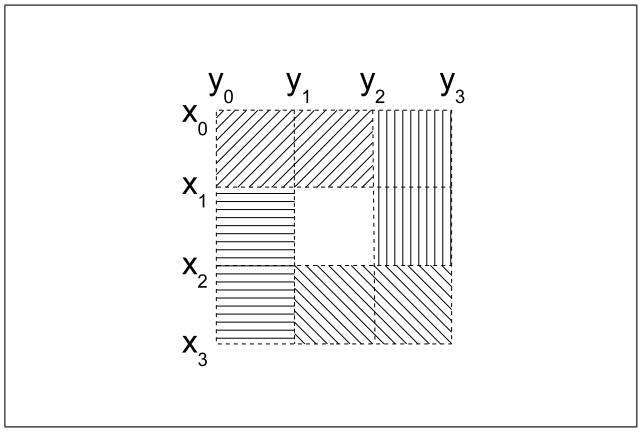

Figure 5: Equidistant layout example.

$x_0, \ldots, x_m$, $m \in \mathbb{N}$, and y-tabstops $y_0, \ldots, y_n$, $n \in \mathbb{N}$, then the set of constraints C on them looks like this:

$$\begin{aligned} C \subset \; \{ \;& a_0x_0 + \ldots + a_mx_m + b_0y_0 + \ldots + b_ny_n = c \\ | \;& a_0, \ldots, a_m, b_0, \ldots, b_n, c \in \mathbb{R}\} \end{aligned}$$

Note that it is possible to use different units for different constraints. A value may be defined in pixels, which makes the actual size dependant on the graphics hardware, or in real-world units like cm, which always produces the same size. In the following we will examine different types of linear constraints, and how these can be solved during the layout process.

4.1 Absolute Constraints

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

$$x_3 = 50.$$

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

$$x_2 - x_1 = 100.$$

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

4.2 Relative Constraints

In contrast to absolute constraints, relative constraints describe the position of tabstops or proportion of areas relative to others. This is useful in order

to adapt the layout to changing circumstances, like UI display size or resolution. The layout engine recalculates the layout when such a change occurs.

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

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

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

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

Since a constraint can contain x-tabstops as well as y-tabstops, it is also possible to specify the aspect ratio of an area. We could, for example, specify the aspect ratio for an area $(x_1, y_1, x_2, y_2, 0, moviepanel)$. Say, we want the ratio of width and height of this area to be 16:9, then we would add the following constraint:

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

4.3 Example

In order to illustrate the dynamic adaption of user interfaces, let us consider the set of areas A of Sect. 3 with the following set of constraints:

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

Using our layout engine, this layout, which we can describe as a tuple $L = (A, C)$, can be rendered with the following shortened snippet of C# code:

```
1   Layout l = new Layout();
2   XTab x0 = new XTab();
3   YTab y0 = new YTab();
4   // ...create all x- and y-tabstops...
5
6   l.Areas.Add(
7     new Area(x1, y1, x2, y2, 0, empty));
8   // ...add all areas...
9
10  l.C.Add(new Constraint(
11          new Tab[]{y1},
12          new float[]{1f}, 50));
13  l.C.Add(new Constraint(
14          new Tab[]{x1, x2, x3},
15          new float[]{-2f, -1f, 1f}, 0));
16
17  screen = l.Render(width, height, context);
```

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 we can see in the screenshots of Fig. 6.

4.4 Constraint Solving

Each linear constraint can be used in order to calculate the position of one tabstop. In order to do that, we first assign each constraint to one of the tabstops it is defined on and then use a Gauss-Seidel fixpoint iteration in order to solve the constraints.

The first part, i.e., assigning each constraint to a tabstop, works a little bit similar to the zero-in topological sorting described in Sect. 3. The problem lies in not wasting a constraint by using it for the calculation of a tabstop that is given by a different constraint already. We solve it by keeping track of the unassigned constraints which could be used to calculate each tabstop that has not been assigned a constraint yet. We choose an unassociated tabstop with a minimum number of assignable constraints and assign one of those constraints to it. If there are more constraints than tabstops, then the layout is overspecified, and we can safely ignore the superfluous constraints.

This is illustrated in Algorithm 2: when the algorithm starts, C contains all constraints and T all tabstops. In the while-loop we assign constraints to tabstops until we have run out of either of them. In line 3 we choose a tabstop *tab* for which a minimum number but at least one constraint is available and assign *con*, one of those constraints, to it in lines 4-5. We remove the tabstop with its assigned constraint at the end of each iteration in lines 6-7.

Algorithm 2 Assignment of constraints to tabstops.

1: $assignment \leftarrow \emptyset$
2: **while** $C \neq \emptyset \wedge T \neq \emptyset$ **do**
3: $tab \leftarrow t \in T$ with $|\text{cons}(t)| = \min_{t' \in T \wedge \text{cons}(t') \neq \emptyset} |\text{cons}(t')|$
4: $con \leftarrow c \in C$ with $c \in \text{cons}(tab)$
5: $assignment \leftarrow assignment \cup \{(tab, con)\}$
6: $T \leftarrow T - \{tab\}$
7: $C \leftarrow C - \{con\}$
8: **end while**

Once we have assigned constraints to the tabstops, we apply the Gauss-Seidel algorithm in order to solve the constraints numerically. This algorithm defines a fixpoint function on the variables, which correspond to our tabstop positions. Without loss of generality, let $x_1, \ldots, x_m$ be the variables representing the positions of the tabstops in our layout, and $x_{i_1}, \ldots, x_{i_n}$, $\{i_1, \ldots, i_n\} \subseteq \{1, \ldots, m\}$, $i_1 < \ldots < i_n$, be the n variables out of the m for the tabstops to which we have assigned a constraint. Of course, although we just call them x_i, $i = 1, \ldots, m$, the variables may stand for a mix of x- and y-tabstops. We initialize all the variables with the tabstop positions calculated using the ordinal information and equidistant scaling, as discussed in Sect. 3, thereby producing our first first approximation $x_1^{(0)}, \ldots, x_m^{(0)}$.

Let the constraint assigned to tabstop x_{i_j}, $j = 1, \ldots, n$, take the form

$$a_{j,1}x_1 + \ldots + a_{j,m}x_m = c_j.$$

Now, we can improve $x_{i_1}^{(k)}, \ldots, x_{i_n}^{(k)}$, $k \in \mathbb{N}$, iteratively by applying

$$x_{i_j}^{(k)} = \frac{c_j - \sum_{i<i_j} a_{j,i}x_i^{(k)} - \sum_{i>i_j} a_{j,i}x_i^{(k-1)}}{a_{i_j}},$$

for all $j = 1, \ldots, n$. For all other x_i the value stays constant, i.e.,

$$x_{i'}^{(k)} = x_{i'}^{(k-1)} \text{ for } i' \in \{1, \ldots, m\} - \{i_1, \ldots, i_n\}.$$

We perform iterations until

$$e = \sum_{j=1,\ldots,n} |x_{i_j}^{(k)} - x_{i_j}^{(k-1)}|$$

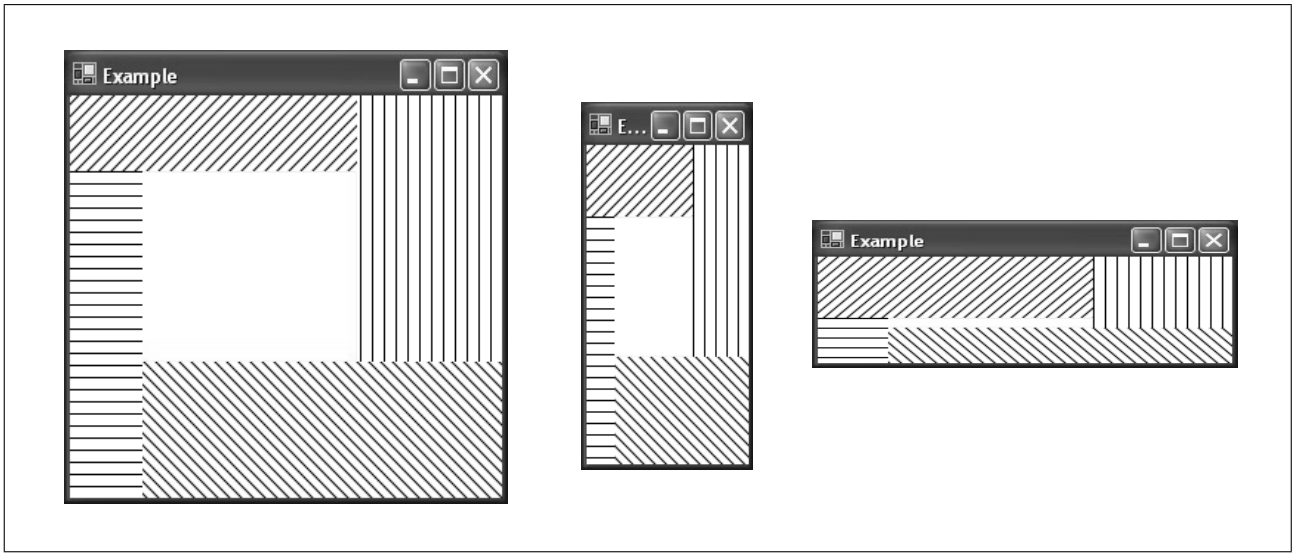

Figure 6: Example layout adjusted to window resizing.

is sufficiently small. More information about this algorithm can be found in (Barrett, Berry, Chan, Demmel, Donato, Dongarra, Eijkhout, Pozo, Romine & der Vorst 1994).

Because the results of the topological sorting of ordinal information is used as initial value, all ordinal properties that are not affected by the linear constraints are retained. Only the positions of those tabstops are adjusted for which linear constraints were given. If the constraints contain conflicts the system will still try its best to render the layout. But, of course, this will at best result in a compromise between the conflicting constraints. Satisfiability of constraints can and should be checked statically, and not while the layout engine is used.

Since the initial values already reflect the basic properties of the desired layout, the constraint solving algorithm converges fast. Precise results are usually achieved in a few iterations. For our example in 4.3 not more than 3 iterations are required. This performance makes it potentially possible to apply this algorithm in real-time systems.

5 Composition and Decomposition

It is possible to compose an UI layout from partial layouts, or extract parts of a layout, thereby creating a new one. This is also called composition and decomposition of layouts, respectively. Different forms of composition and decomposition are conceivable. In this section we want to define a notion of composition that makes it possible to nest tabular layouts into another, and a notion of decomposition that makes it possible to render partial layouts separately.

In order to define notions of composition and decomposition, let us first define the concept of the *bounding box* of a set of areas. The bounding box is a function *bound* that specifies the leftmost, topmost, rightmost and bottommost tabstop of a set A of areas:

$$\begin{aligned} bound(A) \;=\; (& \min_{x \in \text{ xtabs}(A)} x, \quad \min_{y \in \text{ ytabs}(A)} y, \\ & \max_{x \in \text{ xtabs}(A)} x, \quad \max_{y \in \text{ ytabs}(A)} y). \end{aligned}$$

Let $L_1 = (A_1, C_1)$ be a layout, with A_1 being the set of areas and C_1 the set of linear constraints. Furthermore, let $L_2 = (A_2, C_2)$ be the layout and (x_1, y_1, x_2, y_2) the bounding box within that layout that we want to insert into L_1. Let the set of tabstops used in A_1 and A_2 be disjoint, which can always be achieved by substitution. Then, layout L_3, which unites L_1 and L_2 accordingly, is defined as follows:

$$\begin{aligned} L_3 &= (A_3, C_3) \\ A_3 &= A_1 \cup A_2[x_1/x_1', y_1/y_1', x_2/x_2', y_2/y_2'] \\ C_3 &= C_1 \cup C_2[x_1/x_1', y_1/y_1', x_2/x_2', y_2/y_2'] \\ &\text{with} \quad (x_1', y_1', x_2', y_2') = bound(A_2). \end{aligned}$$

L_2 should preferably contain only relative constraints, since absolute constraints may conflict with those of L_1.

Let $L_1 = (A_1, C_1)$ be a layout, and (x_1, y_1, x_2, y_2) the bounding box of the part of this layout that we want to extract. Then, the layout L_2 containing only the extracted part is defined as:

$$\begin{aligned} L_2 &= (A_2, C_2) \\ A_2 &= \{(x, y, x', y') \mid (x, y, x', y') \in A_1 \\ &\quad \wedge x_1 \le x \le x' \le x_2 \wedge y_1 \le y \le y' \le y_2\} \\ C_2 &= \{ \sum_{i=1..m} a_i x_i + \sum_{i=1..n} b_i y_i = c \\ &\quad \mid (\sum_{i=1..m} a_i x_i + \sum_{i=1..n} b_i y_i = c) \in C_1 \\ &\quad \wedge \forall a_i \neq 0, i \in \{1, \ldots, m\} : x_1 \le x_i \le x_2 \\ &\quad \wedge \forall b_i \neq 0, i \in \{1, \ldots, n\} : y_1 \le y_i \le x_2\} \end{aligned}$$

Note that no clipping is performed here. If an area or a constraint is not completely within the extracted bounding box, it will not be extracted.

6 Related Work

In order to create document layouts, most people use simple desktop publishing software like, 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 pre-defined set of properties for each of them. It is not possible to formulate constraints, but

merely possible to change coordinates in an operational manner, like, for example, aligning an element. This means that the layout usually does not adapt well to any changes of the context, like changes of the 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 resizeable 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 to arrange 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 (Walrath, Campione, Huml & Zakhour 2004) represent identical layout concepts. The HTML table construct (Raggett 1996) is a good representative. The colspan and rowspan attributes have been explained in Section 2.1. Furthermore, it features a range of additional table parameters, like horizontal and vertical cell spacing, outset of the table, inset of cells, and parameters for single cells, like padding and cell width. 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, like `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. 2.1, our table concept does not require non-simple cells, but offers superior flexibility by allowing only a partial order of tabstops. It also supports overlapping areas and multiple layers.

The scientific community has examined the concepts of layout already very early. In (Wyk 1981) a graphics typesetting language is described that employs a number of different constraints, also non-linear ones, in order to create images. Although very powerful, solving non-linear 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, like the one described in (Coutaz 1985), support only basic layout schemes, like 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 (Anderson & Sobti 1999), 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 (Borning, Marriott, Stuckey & Xiao 1997, Badros, Borning & Stuckey 2001). 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 on 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, (Beach 1986) 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 (Jacobs, Li, Schrier, Bargeron & Salesin 2003). 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 (Silberhorn 2001) 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.

There are some consistency issues with respect to table definitions, and there is now a good practice on how to deal with these consistency issues. We define a *simple table* as a table that consists only of *simple cells*, i.e., cells that do not make use of colspan or rowspan. We can create a corresponding simple table for a table that uses colspan and rowspan by assigning to each cell using rowspan r and colspan c a set of cr simple cells. A syntactically correct table definition is *overlap-free* if the simple cells assigned to cells spanning over multiple rows or columns do not collide. A table is *complete* if all simple cells completely cover the table. Completeness is generally not prescribed. From the intended functionality, typically only overlap-free definitions would be considered correct, but it is good practice that renderer like HTML browsers try to salvage incorrect page specifications by writing cells on top of each other. Note that this kind of overlap is different from the layering introduced earlier.

6.1 Further Work

Our approach specifies layouts on a low but simple level. It would be interesting to examine higher-level abstractions for layouts or for parts of them. In (Jacobs et al. 2003, Jacobs, Li, Schrier, Bargeron & Salesin 2004) abstractions for the layout of whole pages, so called templates, are discussed, and algorithms that automate the generation of documents with various page layouts and formats from sets of

templates and streams of content. It would be interesting to explore similar notions for the generation of user interfaces. The idea of abstract UI specifications that can be rendered in different contexts, e.g., on different devices, has already been examined, but approaches are usually limited with regard to layout. In most approaches, e.g., in (Grundy & Yang 2003), layout is based on the concept of a simple GridBag (Walrath et al. 2004).

Our research implementation is a standalone layout engine. One could consider to build a best-effort preprocessor for conventional GridBag layout approaches, for example a tool that maps the layout into a HTML table in a best effort approach. Such a preprocessor would be helpful for bringing some advantages of the new layout to conventional technologies like HTML. However one uses a lot of flexibility especially in the resizing functionality. The task of the best-effort preprocessor helps however to illustrate the deficiencies of the other approaches. take an example similar to Fig. 1 but with n resp. m unrelated columns in the two types of rows. The preprocessor has to compute the complete rendering in order to create the correct total ordering of the tabstops, and encode this into colspan attributes. Hence in this example, the HTML renderer that will later interpret the HTML output provides no separation of concerns, or no partial solution of the problem; anything hat to be done beforehand. From a software engineering point of view this shows how serious a problem such a deficient interface can be.

Our approach has added benefits for text processing and document layout in general. Text processors offer the possibility to set tabulators fro a special paragraph type and to reuse this paragraph type at different positions in the text. In these approaches the concept of tabulator is however mostly completely separate from the concept of tables. Our approach allows to unify these concepts, so that tabulators become tabstops. As a footnote of this, our approach allows the use of tabstops for defining more general positioning elements like decimal tabulators. Moreover, other major concepts of document layout, like flowing and pagination, can be directly translated into concepts about areas in a tabular layout. Many challenges in the generation of optimal layouts can be seen as problems about the wrapping of areas, i.e., the distribution of a list of areas into a set of slots. Words, lines of text, paragraphs and text columns can be interpreted as hierarchically structured tables.

7 Conclusion

We identified the shortcomings of very widespread types of tabular layout, like the HTML table construct and other grid-based tabular layouts. These approaches rely on a total order of graphical elements and are therefore not flexible enough in some cases. Our approach uses rectangular areas that are aligned by a partial order of horizontal and vertical tabstops and adapts more efficiently to content and varying context parameters, like screen size. We think that the concept of areas captures the topology of a layout in a very natural and intuitive manner, and we suggest the use of linear constraints for the specification of additional metric information. Those constraints usually have a very simple form and therefore fit well with the concept of areas. We described notions of layout composition and decomposition, and a fast algorithm for rendering layouts based on areas and linear constraints. We implemented the algorithm in a layout engine for C#, which can be downloaded at `http://www.cs.auckland.ac.nz/~lutteroth/projects/layout/`.

References

Anderson, R. J. & Sobti, S. (1999), The table layout problem, *in* 'SCG '99: Proceedings of the fifteenth annual symposium on Computational geometry', ACM Press, New York, NY, USA, pp. 115–123.

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

Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C. & der Vorst, H. V. (1994), *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*, SIAM, Philadelphia, PA.

Beach, R. (1986), Tabular typography, *in* 'Text Processing and Document Manipulation', The British Computer Society, Cambridge Univerity Press, pp. 18–33.

Borning, A., Marriott, K., Stuckey, P. & Xiao, Y. (1997), Solving linear arithmetic constraints for user interface applications, *in* 'UIST '97: Proceedings of the 10th annual ACM symposium on User interface software and technology', ACM Press, New York, NY, USA, pp. 87–96.

Coutaz, J. (1985), 'A layout abstraction for user-system interface', *SIGCHI Bull.* **16**(3), 18–24.

Gross, J. L. & Yellen, J., eds (2003), *Handbook of Graph Theory*, CRC Press.

Grundy, J. & Yang, B. (2003), An environment for developing adaptive, multi-device user interfaces, *in* 'CRPITS '03: Proceedings of the Fourth Australian user interface conference on User interfaces 2003', Australian Computer Society, Inc., Darlinghurst, Australia, Australia, pp. 47–56.

Jacobs, C., Li, W., Schrier, E., Bargeron, D. & Salesin, D. (2003), 'Adaptive grid-based document layout', *ACM Trans. Graph.* **22**(3), 838–847.

Jacobs, C., Li, W., Schrier, E., Bargeron, D. & Salesin, D. (2004), 'Adaptive document layout', *Commun. ACM* **47**(8), 60–66.

Raggett, D. (1996), 'RFC1942: HTML Tables'.

Silberhorn, H. (2001), Tabulamagica: an integrated approach to manage complex tables, *in* 'DocEng '01: Proceedings of the 2001 ACM Symposium on Document engineering', ACM Press, New York, NY, USA, pp. 68–75.

Walrath, K., Campione, M., Huml, A. & Zakhour, S. (2004), How to use GridBagLayout, *in* 'The JFC Swing Tutorial: A Guide to Constructing GUIs', Addison-Wesley Professional.

Wyk, C. J. V. (1981), A graphics typesetting language, *in* 'Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation', pp. 99–107.