Automatic Application of Layout in Coordinate-based Interfaces during Localization

August Max Seelemann

July 8, 2010

With software localization becoming more important to successful software development, concise tools that aid this process are needed. However, in systems where interfaces are based on static coordinates, localization imposes a huge problem with its need for different sizings for different languages. After an analysis of existing interface layout concepts, we describe the Auckland Layout Model, an instance of which we will generate from given coordinates only. Using the generated model, we construct an algorithm that is able to cope with changing sizes of interface elements and rearranging them accordingly. The algorithm is specifically suited for changes occurring during localization. Finally, we embed our method into the Localization Suite, an integrated set of localization tools for Mac OS X.

Contents

Chapter 1

Introduction

For any (software) product to be internationally successful, its developer is required to adjust it to various regional and user-specific habits and settings. This process is called "internationalization" [1] and includes specifics like currency, number and date formats, the writing direction, and—most importantly—the language. The result of translating an application into a specific language is commonly referred to as "localization". It is common for a software product to have ten, twenty or even more localizations, that need to be kept synchronous. While there are many issues that may arise during localization, the most prominent one is the fact that strings in different languages have different lengths and thus need a different amount of space for display.

A word or a sentence might be much longer or shorter in one language than in another. For instance, English, which is usually the development language, is much more compact than others. Thus, most localizations will have longer strings. The result is that elements of the user interface might need more space than in the original localization. A button might be sized to fit exactly its title "Edit" in English, but also needs to accompany "Bearbeiten" in the German localization, which is more than twice as long. The sizing and placing of interface elements in a window is an important matter, but dependencies between these elements must be considered as well. For example, in a window with multiple buttons for different actions, these buttons should all be of the same size. These restrictions and requirements are generally referred to as "constraints".

In modern application frameworks, the most common approach to solve this issue is to have an abstract description of how the user interface should look like. Such a description specifies where each element is placed, the relations between some elements and how each element should behave under various conditions. Based on this specification, the layout can be generated dynamically for each localization, thus solving the issue of different lengths by design. Many well-known systems use this method—examples are Java Swing [2], Trolltech's QT [3], Microsoft's Windows Presentation Foundation [4] or Google's Android [5]. All these systems use a diverse set of so-called "layout managers", classes that implement such an abstract description. They will be analyzed in more detail in Chapter 2.

The nice attributes of layout managers come at a certain cost—their complexity in large scenarios. This complexity not only includes their possibly complex structure, but also the complexity of creating or modifying them. Also, for some scenarios, a layout manager might not be feasible. This is (part of) the reason, why other platforms like [6] do not follow this approach. Instead of using a description, elements are placed directly on the window. Concrete coordinates are directly assigned to the controls, a task that would otherwise be done by a layout manager. Using a visual tool like [7], it is now very straightforward to create a user interface. Static coordinates are simple to understand, as they match exactly what the creator had in mind. Prototyping and creating an interface can be done very quickly as it does not involve any abstractions. This approach, of course, also has its problems: namely, that localization proves to be a big issue.

When loading a different language, the translated strings may no longer fit the control sizes of the original layout. When using a layout manager, this is no issue at all. Using its abstract description, controls will be laid out in a way that the strings always fit. However, with static coordinates there is no layout manager. There is no abstract description that can be used to re-size or re-arrange the controls. There is no immediate possibility to adjust the interface.

A proposed solution [8] is to create a copy of the original interface, inject the translated strings and manually adjust the sizes and positions. For example this can be done by opening each localization in the visual tool and adjust the sizes as needed. Again, this solution imposes several problems. It is time-consuming, thus expensive, and also error-prone to manually edit hundreds or even thousands of interfaces. In addition, the approach requires subsequent changes in the original interface to be transferred to the other localizations. Luckily, this does not need to be done manually. Tool support for solving the problem of updating changes exists, which will be discussed in detail in Chapter 4.

What remains as an issue is the need to manually resize each interface for each language. Until now, we do not know of any published solution trying to solve this issue transparently. In this paper, we introduce an algorithm that can automatically resize user interfaces using only an original interface and the translations. We then embed this algorithm into the Localization Suite [9], a suite of tools aided to extend the automation in maintaining multiple localizations on Mac OS X.

The rest of this paper is structured as follows: starting with Chapter 2, we give an overview of existing interface layout techniques, followed by a brief introduction to the Auckland Layout Method [10]. In Chapter 3, we introduce an automatic recognition from a coordinate-based interface only as well as our set of recognized constraints. We then describe the application of the layout during a localization in Chapter 4. Chapter 5 follows by describing the process of interface localization in Mac OS X [11], as well as a suite of tools aiding this process. We embed our algorithm into these processes, also presenting what is needed to allow incremental changes. Chapter 6 compares our work to related ones. We give a summary and your conclusions in Chapter 7.

Chapter 2

Background

This chapter gives a general overview of user interface layout using layout managers, coordinate-based interfaces and an introduction to the Auckland Layout Model (ALM) [10], which serves as a basis for our work.

2.1 Layout Managers

A layout is an abstract description of how the elements in a user interface (for instance, in a window) are arranged. Figure 2.1 shows an example of a window. Laying out a user interface is the process of applying the layout on the elements to generate concrete coordinates. In the figure, one can see the bounding boxes, i.e., coordinates, of each element in the interface. The positioning of the elements can be described in an abstract way.

As already mentioned in Chapter 1, many systems use so-called "layout managers" to implement this abstract description. In our example, we would use one that is capable of arranging controls next to each other. Usually layout managers can be nested. For example, the manager describing the positioning and sizing of the two buttons would be embedded in a layout manager which divides the window into an upper and a lower part.

Most of these layout managers can be generalized as a table. A table consists of rows and columns. Each element is assigned one cell, the intersection of a row and a column. In the example, the two buttons "Cancel" and "Yes" could

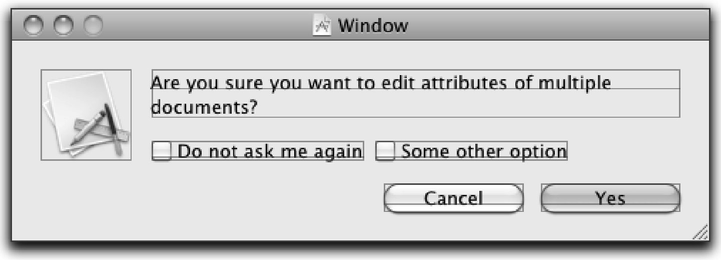

Window
Are you sure you want to edit attributes of multiple documents?
Do not ask me again
Some other option
Cancel
Yes

Figure 2.1: An example window showing the layout bounding boxes of all controls.

be placed in a table consisting of only one row and two columns. The layout manager might also be able to define that the cells must be equal in size.

While the implementation of a layout heavily depends on the interface system, the basic process is always the same. First, the abstract description is formulated as a set of layout managers. Then, the strings and contents are put into the interface elements. Some constants may be generated in this process, like the fixed size of an icon. Finally, the concrete coordinates of all elements are computed and put into place.

Layout managers are universally usable for both static and dynamic layout. Static layout is the initial arrangement of interface elements. This takes into account any settings the developer has made and produces a set of coordinates. Static layout also covers localization. Dynamic layout, however, refers to the adjustment of an interface during runtime. For example, when the user resizes the window or the contents of some controls change, the interface may need to be adjusted. When using a layout manager, this can be done by simply re-applying the procedure for the static layout with the changed settings. The usage of the same layout manager for both static and dynamic layout is not required in general. Some systems (like the one described in the next section) have separate mechanisms for dynamic layout.

2.2 Coordinate-based Interfaces

As mentioned in Chapter 1, not all systems for developing user interfaces have support for layout managers. For example, in Apple's Application Kit framework, the only way to layout a user interface is the direct assignment of coordinates. Using a visual tool [7], the developer places the controls directly. This promises to be a fast method for prototyping and developing interfaces. Nearly any arrangement can be created in a straightforward way. However, this approach has the problem that the coordinates are static. Replacing strings during localization might require some elements to resize in order to display the whole content—which is inherently not possible if coordinates are fixed.

There are three proposed solutions to this problem:

- The default solution is to create a copy of the interface and resize it manually to fit the contents. For each localization, each interface is copied, adjusted and then shipped in the product. Creating these copies and adapting them to subsequent changes has tool support [12]. Despite that, the resizing has to be done by hand which can be time-consuming and error-prone.

- Another approach is to initially size all controls in a way that they fit all possible translations. While this approach is known to be used in praxis [13], it may hinder interfaces from being optimal for each language. Elements may take up more space than needed in all localizations because of a single language with high size requirements.

- One could also try to implement layout managers to the system, which would partially or completely replace the static coordinates by an abstract description. We know of no public evidence that something like this has been implemented. If such a system would exist, it would probably not be much different from other layout managers.

All these solutions are not optimal, as they all make huge compromises. The first approach has the drawback of the costs to create and maintain it, the second with the quality of the outcome. The third approach requires developers to write additional code just for localization. The question that comes up is whether a solution that has none of these compromises and can be applied completely automatically exists, maybe with a little effort to set it up once.

On a side note, these problems refer to static layout only. For dynamic layout the AppKit has an approach called "struts and springs" [14]. Each interface element has, for each border and each direction, a resizing mask attached. This resizing mask can be either fixed or dynamic. If a control has a fixed mask, this property will stay the same throughout the complete dynamic sizing. If a mask is dynamic, the property will change relative to the change of the interface. For example, if a control has a dynamic width and the interface increases by 20% in width, the control will as well. When having a look at our example from above, the two buttons in the lower right corner will have a fixed lower and right border, as well as fixed width and height. The description text on the top might have a flexible width and fixed borders on all sides.

The decoupling of static and dynamic layout is another advantage of this approach. Both are very simple to use and easy to understand. The issue of static layout remains, however. Having one interface for one language and adding a different language still causes the need to resize controls. Using the dynamic layout system is no option, as it refers only to the complete interface and not to single controls. Using the dynamic information to adjust the static layout is not possible either: If a window cannot be resized by the user, the dynamic layouting will never be performed. Thus, it may contain any arbitrary information and must not be used.

In order to adjust the static layout of a coordinate-based interface, an abstract model describing this layout is absolutely required. In our approach (see Chapter 3), we generate a simplified version of the Auckland Layout Model.

2.3 The Auckland Layout Model

The layout of user interfaces can be described using a table. A table is a set of rows and columns, whose intersections are called cells. Each cell may be either empty or contain any kind of content, like a picture, some text, or—in the case of user interfaces—controls. Cells may also span multiple rows or columns, adding more flexibility to the approach.

Without going too much into detail, the problems in a classical table approach, existing in merely all layout manager-based systems, should be outlined. While not being used in user interfaces, a well understood model of tables comes from HTML [15]. Figure 2.2 shows an example of a table describing the interface from Figure 2.1. Each element of the interface is placed in a cell spanning one or more rows or columns.

As can be from Figure 2.2, representing the fairly simple example from above using a classical tabular approach may quickly become complicated. With only six elements in an interface, any horizontally overlapping cells need to span multiple columns. This is by far not optimal when seen from a simplicity point of view. For example, Cell 4 needs to span three columns (3, 4 and 5) to have its location defined. Cell 6 needs to span two columns (5 and 6) as well.

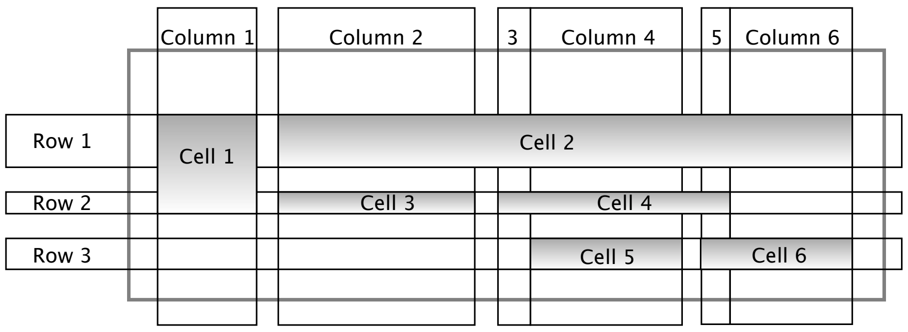


Figure 2.2: The layout of the controls, represented as cells, described using a tabular approach.

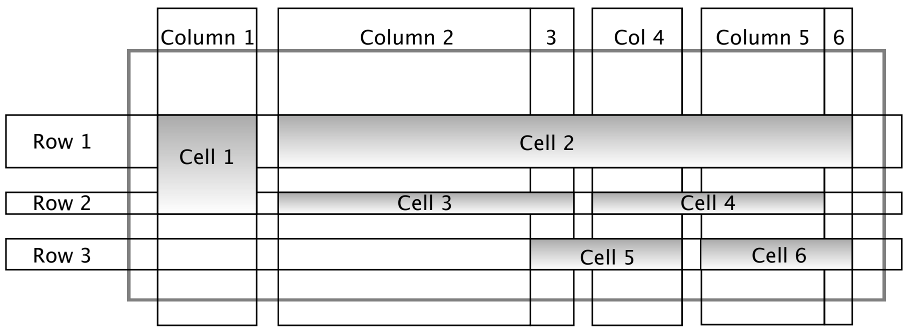


Figure 2.3: Cell 3 increased in size, requiring the tabular layout to change its structure.

Even worse, the table might also add unwanted dependencies between actually independent elements. Column 5 is required and used by two cells (5 and 6), while their layout is actually completely independent.

These unwanted dependencies create more problems: Figure 2.3 shows an example, where the contents of Cell 3 changed during localization, requiring the cell to increase in size. From a natural understanding (as in Figure 2.1), the control in the cell would adjust its bounds, moving the element in Cell 4 to the right. In the tabular approach this requires the structure of the columns to change. Instead of using just one column, Cell 3 has to span two columns (2 and 3), similar observations can be made for Cells 4, 5 and 6.

The Auckland Layout Model (ALM) [10], does not suffer from these problems. It can be seen as a generalized table. The basic idea is to not have any predefined rows and columns, but instead use tab stops to delimit the bounds of a cell. A tab stop in turn represents a position. Each cell is then assigned a minimum and a maximum horizontal and vertical tab stop. These tab stops are not in a total ordering, as the columns in a classical table need to be, but only define a partial order. Using this weakened requirement, no unnecessary or even unwanted dependencies are introduced between cells. If two tab stops

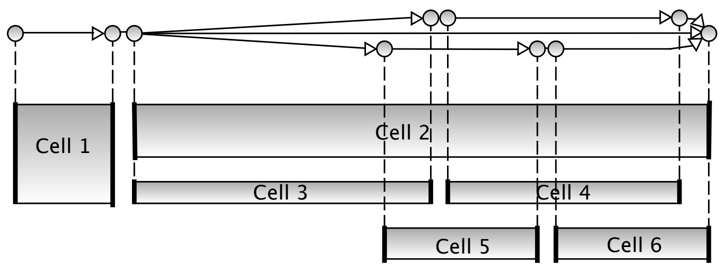


Figure 2.4: The representation of the interface using the Auckland Layout Model (ALM). The tab stops form a partial ordering, visualized using a directed acyclic graph. For simplicity reasons, only vertical tab stops are shown.

are not comparable with regard to the partial order, they can be positioned independently. Only comparable tabs must stay in the defined order. Figure 2.4 illustrates a possible representation of the discussed interface using the ALM.

As can be seen from Figure 2.4, a change of Cell 3's size will affect its maximum vertical tab stop only. Under some circumstances, Cell 4's tab stops will have to move as well, but there is no effect on the tab stops of Cell 5 or 6, respectively.

In order to define more cell attributes than just their relative position, and thus defining more attributes of the cells than just their relative position, the model has support for arbitrary linear constraints. For example, some elements of the interface might have a minimum size (defined by their contents, i.e., text or label). These minimum requirements can then be expressed by linear inequations. An example is $x_2 - x_1 >= 100$, where x_2 is the right and x_1 the left tab stop of a cell. Some controls like the two buttons in the example may have a fixed distance between them. An equation for this distance would hence be defined. Other constraints could fix the position of a tab stop or might define a specific area to be thrice the size of another and so on. The approach is very flexible and basically allows all kinds of linear dependencies being expressed in a straightforward way.

The complete model can be transferred into linear (in-)equations, which can then be passed to a solver that computes positions for all tab stops accordingly. One such solver is *lpsolve* [16], which is also used by the ALM. Controls are positioned according to the positions of their tab stops. The resulting layout will be a minimal solution fulfilling all constraints.

Chapter 3

Layout Generation

When creating user interfaces using layout managers, the developer not only creates an interface, but an abstract description of the layout as well. This description can also be used during localization. Its ability to handle varying lengths of texts is inherently linked to the working principle of layout managers.

However, when creating user interfaces statically by assigning elements coordinates directly, no such description is created. Of course, the interface still has some kind of layout. The elements are still somehow arranged in relation to each other. But this layout or an abstract description of it is never formulated and written into source code like with a layout manager. When replacing texts during localization, interface elements may no longer be able to show all contents, thus the interface needs to be rearranged. But rearranging needs some kind of layout description. Hence we need a method to automatically generate a layout description for a given coordinate-based interface.

Such a method would have to extract all information, especially the structure and the constraints, from the coordinates only. That said, it is nearly impossible to recognize what a developer may have had in mind or not. Each applicable pattern of whatever kind might match intentionally or by accident, bearing a high risk of taking the wrong guess. Therefore, recognizing arbitrary relations between the elements is simply not feasible.

We can use the fact that the model to be generated is not needed to create an initial interface but that we can change an already existing one. There already is a statically laid-out version of the given interface and that needs to be adapted to a different language only. The optimal solution is the original that was created by the developer. The target is an adjusted layout that fulfills all constraints and is as close to the original as possible.

In this chapter, we introduce an algorithm that reaches for this target. In a compact view, it first generates an ALM-like model of the layout, recognizes a few very specific constraints and then does a specialized application using linear programming. We chose the Auckland Layout Model due to its great flexibility. Its inherently given ability to compose nested tables into a single one comes at a great benefit for us. The advantages over other tabular layouts (see Chapter 2) contributed to our decision as well.

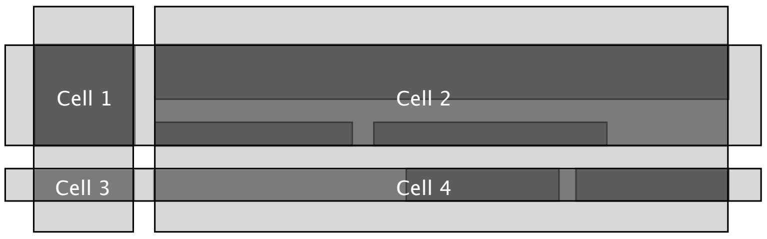


Figure 3.1: The row and column grouping applied on the interface from Figure 2.1. The dark shades of gray are the original bounding rectangles (frames) of the interface elements. The light shades of gray represent rows and columns. The intersections of rows and columns are the cells.

3.1 Recognition

The first step in generating our (ALM-like) layout model is to create its structure. The structure is mainly defined by the tab stops and the partial ordering amongst them. Each control will then be assigned a cell between four such tab stops (one for each border).

As discussed in Chapter 2, the ALM supersedes a regular tabular layout. Every table can immediately be transformed into an ALM, while the other direction is not always possible[1]. When excluding a few special cases, almost every user interface can be expressed as a tabular structure. Using this fact, we describe a recursive algorithm that works by recognizing nested tables. These tables are merged into a single instance of our layout model afterwards. Throughout the description of the algorithm, we are using the same example as in the chapter before, the interface as shown in Figure 2.1.

The main idea is to group the bounding rectangles (also called frames) of the interface elements. Seen from each direction (horizontally and vertically), all overlapping frames are reduced to a single rectangle. Extended to the borders of the layout, these rectangles represent rows and columns. The intersection of a row and a column forms a cell, containing a varying number of interface elements.

Figure 3.1 shows an example. There are six elements in the interface, represented by their six bounding rectangles (dark gray). These rectangles are—once horizontally, once vertically—grouped, forming two columns and two rows (light gray). The cells are the intersections of rows and columns. Together, rows, columns and cells form a primitive table, giving the layout a raw structure. Now the very same approach is applied recursively, creating the nested tables previously mentioned.

After performing a grouping step, each cell may contain a varying number of elements. Intuitively, multiple overlapping elements are grouped into one column or row, thus creating the possibility of more than one element being in a cell. The aim is to have exactly one element in each cell, which can be achieved by nested grouping steps. There are three possible cases distinguished by the algorithm:

[1]For example, we do not know of any tabular layout that supports overlapping cells.

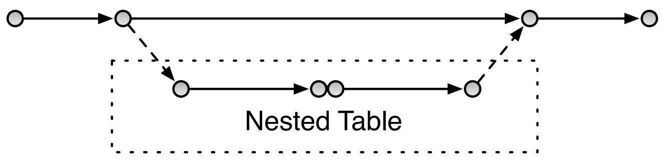


Figure 3.2: Nesting a tab stop graph from a nested table into an outer tab stop graph.

1. A cell does not contain any elements. Therefore, it is not important for the layout and can safely be ignored. In the example of Figure 3.1, this applies to Cell 3.

2. A cell contains exactly one element, and is therefore optimal. It can be added to the final layout. This applies to Cell 1.

3. A cell contains more than one element. The grouping algorithm is applied recursively on the elements in the cell only. The cell is then replaced by the resulting nested table. This applies to Cell 2 and 4.

For the generated rows and columns, a linear sequence of tab stops is generated. Each end of a row or a column is represented by exactly one tab. The result is a set of adjacent tab stops for each direction. Each cell with exactly one element is added to the layout and assigned one tab stop accordingly for each border. Note that these tab graphs have no branches. The reason for this is the simple flat tabular structure of the table recognized by the grouping. Tab graphs will gain branches as soon as nested structures are merged into them.

When a cell generated by the grouping contains more than a single element, it needs to be split up. According to rule 3 above, the grouping algorithm will be performed again, but this time on the elements in the cell only. The result is a tabular model of the elements inside the cell, an instance of our layout model. Figure 3.2 illustrates how this layout model is then merged into the outer model. Because layout is defined only by the two tab stop graphs and the cells, merging is pretty simple. Both the horizontal and vertical tabs are just placed in between the according tab stops of the originally generated cell.

Using the previously discussed example, Figure 3.3 depicts a complete run of the grouping algorithm on this interface, split into six steps: Step 1 starts with the complete interface, applies the grouping algorithm and generates the outermost table. This is exactly the process described above that can also be seen in Figure 3.1. Figure 3.3 shows both the horizontal and the vertical tab stop graph, with the rectangles of the columns highlighted in light gray. The algorithm found two rows, two columns and hence four cells. The cell containing Element 1 only is already optimal and can be added to the layout. The upper right cell (Cell 2 in Figure 3.1), however, contains more than one element (more precisely, Elements 2, 3 and 4). For this cell (Step 2) and (Step 4) for the lower right cell (Cell 4 in Figure 3.1) the grouping is performed again.

In Step 2, the upper right cell from Step 1 containing Elements 2 to 4, is split up again. The grouping finds only one column, but two rows. The upper cell now consists of Element 2 only, and can be added to the layout. The lower cell, in contrast, must be split again, resulting in Step 3, which generates two cells—one for Element 3 and one for Element 4. Similarly, Step 4 splits up the

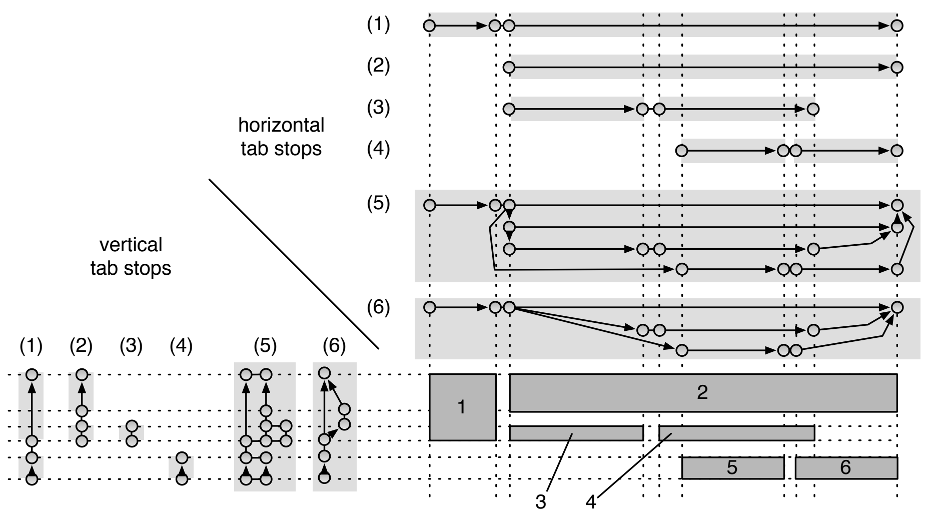


Figure 3.3: A complete run of the grouping and tab stop generation. Steps (1) to (4) depict the (recursive) calls, represented by the generated graphs as well as the found rows/columns (light gray areas). Step (5) illustrates the complete, composed tab stop graphs, which have been reduced again in Step (6).

lower right cell generated by Step 1 (which contains the Elements 5 and 6) and splits it up into two cells as well.

Alongside these recursive groupings, tab stop graphs for each direction are generated and merged into each other. The result of these merge operations (see Figure 3.2) is shown in Step 5, with both the complete graphs for each direction. The tabs of Step 2 will be merged between the tabs of the upper right cell of Step 1. The stops from Step 3 will be merged between the stops of the lower cell of Step 2, because it was called recursively from there. Similarly, the tab stops of Step 4 go between the ones of the lower right cell from Step 1.

The merged graph contains lots of redundant tab stops. One example is in the horizontal graph, where two tab stops share the rightmost position. Another example might be the parallel edges between two stops. Step 6 shows the result of a very simple reduction: all directly adjacent tab stops which have the same position are merged into a single one. In our example, four tab stops are removed from the horizontal and six from the vertical graph. This reduction makes sense, as duplicate tab stops are likely to be created during each recursive grouping step.

The result of the complete grouping algorithm is a flat, tabular, ALM-like model of the layout of the interface. It contains two tab stop graphs (one for each direction) and a cell for each element. This has been achieved by recognizing nested tables that are composed into a single one afterwards. Note that the outcome of the grouping is just the structure and lacks any kind of sizing information or constraints.

3.2 Constraints

As pointed out in the last section, the result of the grouping algorithm is the structure of the layout model only. Except the original positions of the tab stops there are no constraints generated. In the Auckland Model, the developer has the ability to describe any kind of linear dependencies and constraints between any set of tab stops. In the example of the previously used interface from Figure 2.1, the developer might want the big text label to be six times the width of the icon on the left. This is easily expressed using an equation of tab stop positions, but very hard to recognize using an algorithm. The problem is the uncertainty that comes along with such a recognition. Any speculation also introduces nondeterminism to the whole process, which is unwanted. The chance of high error rates actually turns out to be a real problem.

Instead, we decided to only rely on very determined, very basic principles in user interface layout. As for now, our system supports only two very very determined and basic, yet powerful, types of constraints and their automatic recognition.

The first type of constraint refers to cells. When applied, it states that some set of cells should have the same width or height. Therefore we call it *size constraint*. The applicability of this constraint becomes immediately visible when looking at the interface in Figure 2.1: The two buttons on the lower right corner have the same size. If one of them has to increase its width due to a string, which does not fit, the other button should be resized as well. The main purpose is to ensure symmetry. Of course this constraint is recognized by our method. More specifically, each combination of cells is checked for the satisfaction of the following two criteria:

1. The cells must have the same dimension. (Our current implementation checks for width only, but may easily be extended to check for height as well. We made this simplification as most controls like buttons have a fixed height anyway.)

2. The cells must contain similar elements. Similarity is defined in terms of the controls and may be simply described as "the same kind of control". A button is similar to a button but not to a text label.

Taking the example from Figure 3.3, the algorithm would now add a size constraint for the cells containing Element 5 and 6 to the layout.

The second constraint, the *distance constraint*, is also very simple. The distance between any two tab stops can be constrained to a minimum and a maximum. When minimum and maximum are the same, the distance is considered as fixed. As previously, while the model (in this case the constraint) applied to a lot of usage scenarios, the recognition algorithm is much more restrictive. For a distance constraint to be recognized, several conditions must be met:

1. The two tab stops must be directly adjacent.

2. Both tab stops must be used by at least one common cell.

3. There must be a gap between them. More precisely, the smaller tab stop may not have any cells in the larger direction and the larger tab stop may not have any cells in the smaller direction.

If we apply these three conditions to the example in Figure 3.3, three distance constraints for the horizontal direction and two for the vertical direction would be recognized. In the horizontal case, the first fixed distance will be between the tab stop to the right of Element 1 and the one left of Element 2 or 3, respectively. These two stops are adjacent, have both at least one cell using them and share a gap. The next ones, between the cells of Elements 3 and 4, and between the cells of Elements 5 and 6, are basically the same. Please note that the "gap" is only defined in terms of the cells attached to the concerned tab stops. To emphasize the functional principle of the third rule, another example should be analyzed: One might expect that the distance between the right tab stop of Element 4 and the right tab of Element 2 might also be fixed. Both tab stops are adjacent, have cells using them and no cells in between them. However, while the tab stop on Element 4 has no cell in the larger direction, the stop at Element 2 has one in the smaller direction—which is illegal according to rule 3. The cell containing Element 2 is still "between" the tab stops in terms of that rule.

When it comes down to determining the distance in a distance constraint, the original position is concerned. Small distances below 10 units are fixed, others are flexible with 10 units as the minimum. This heuristic that works well in the scenarios we have tested up to now. On different platforms, different design principles may exist, and so the heuristic might need to be different.

3.3 Discussion

Analyzing the algorithm in detail reveals some observations that are important to discuss. When creating it we made some simplifications that may prove to work or to be too simple. These are the most important aspects:

- All elements must always lie within a cell. No element can lie in more than one cell or partially outside of a cell—otherwise the grouping would have created different cells, that again would contain the bounding rectangles of these elements.

- The algorithm may never terminate if one special case is not excluded. Under some circumstances, it will not be possible to split a cell containing multiple elements into multiple cells. For the examples given in Figure 3.4, the algorithm will always return only one row and one column and thus recurse for ever. In our implementation, the problem is circumvented by treating the contained elements like a single one with a fixed size. While theses elements are still part of the model, their contents can only be completely ignored. Additionally, in order to work around related issues, each cell in a cell constraint must contain exactly one element.

- A cell's bounding rectangle does not need to be of the same size as its control's bounding rectangle. In contrast to the original ALM, our model allows elements to have a position and an inside inside a cell. When looking at the Figure 3.3, removing Element 6 will result in such a case for Element 5. The created cell will be as wide as Element 2 and hence much wider than the button. While it may sound more complicated at first sight, having cells independent from the actual elements makes the

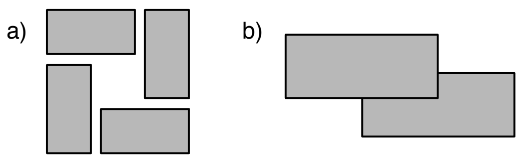


Figure 3.4: Examples for arrangements of elements that cannot be derived into a tabular structure by the grouping algorithm. a) Elements arranged in a way that no column or row can be separated by a grouping. b) Elements that overlap, and thus cannot be split into rows or columns.

approach more flexible and tolerant to small deviations. One example would be a list of text labels, all arranged on top of each other, each with a different width. When applying the grouping algorithm, the returned cells for these elements will have the same width, with the same left and right tab stop. The resulting model is not only simpler, but also the dependencies in layout between the labels are expressed accordingly.

- Of course, the algorithm can create false positives, i.e., detect some dependencies or generate constraints that may not be wanted. One approach to cope with this would be to improve the algorithm such that the results are better, which certainly will happen. Besides that we have foreseen a way to allow the user (i.e., the developer) to manipulate the model after it has been generated. Realizing this is in the focus of our future work.

- Another problem may be controls with dynamic content. Let's say we have a dialogue that is used across multiple applications and in every application it displays some different text. The content of some of its controls would then be generated at runtime. Obviously, there is no way to extract this information from the static interface only. We cannot make any assumptions about what the content may be or not. This problem is inherent to our approach of calculating static layouts only.
  A good workaround would be to fill the control with dummy text that resembles the size of the content generated at runtime. The texts would also be translated (putting a little overhead on localization) and then also put into the interface for static resizing. Because it resembles the sizing of the actual dynamic content, the layout mechanism will create a somewhat appropriate result. In cases where this is not feasible, the interface would then need to be adjusted manually or during runtime.

One additional note: the limitations presented here are mostly limitations of the recognition algorithm and not of the model. The algorithm can be extended to handle these limitations. However, as we have not yet experienced any problems with our method in practice, these extensions are beyond this work. Most interfaces do not seem to contain our corner cases.

Chapter 4

Layout Application

Layout Managers like the ALM are used to layout a given interface from scratch, assigning each element a position. However, the model we generate for a given interface is not complete enough to perform such an application. There are too few constraints and no formalization sizes other than distance constraints. Using a traditional application we would create a telescoped interface where all elements will be sized to their minimum and moved as close together as the constraints allow. In this section we deal with the process of applying changes to elements using a generated model.

4.1 The Application Mechanism

Knowing the original tab stop positions allows us to define our outcome in relation to the original. The best possible result we can achieve is in fact the original, which happens when all texts fit. Otherwise the interface is adjusted, with no need of creating it from scratch. Designing a method that applies changes is very sufficient. Notice that this is not possible using a traditional layout manager. This approach requires a previously laid out interface.

In the original Auckland Layout Model, the application of a layout is the solution of a set of linear inequations. In our case, as we have an additional notion of optimality, our application can be formulated as a linear optimization (also called linear programming) problem [17]. A linear optimization problem consists of two parts: a set of linear (in-)equations and a linear objective function. The target is to minimize (or maximize) the value of the objective function under the constraints of the inequations. It can be formalized as follows:

$$\begin{aligned} min: c_1 * x_1 + c_2 * x_2 + \ldots + c_n * x_n & & \qquad (4.1)\\ c_{11} * x_1 + c_{12} * x_2 + \ldots + c_{1n} * x_n &\leq k_1 \\ c_{21} * x_1 + c_{22} * x_2 + \ldots + c_{2n} * x_n &\leq k_2 \\ &\vdots \\ c_{m1} * x_1 + c_{m2} * x_2 + \ldots + c_{mn} * x_n &\leq k_m \end{aligned}$$

In summary, a linear optimization problem consists of n variables (x_1 to x_n), a linear objective function and m linear (in-)equations. After solving the problem, the result is a set of values for all x_i such that the objective function is minimal with respect to the constraints. We decided to employ a tool called *lpsolve* [16] which implements the simplex method [17].

In our situation, the variables x_i will represent the positions of the tab stops. We are able to express our complete model using linear inequations only. Beginning with the tab stop graph, each component needs to be encoded. For each two adjacent tab stops, represented by their positions x_i and x_{i+1}, formula (4.2) is the resulting inequation.

$$x_i \leq x_{i+1} \tag{4.2}$$

As for all linear inequations, this inequation can be transformed into a normal form like in (4.1). Basically, all variables appear a single time on the left side, while all constants are summed up on the right side. These normal forms are required by some tools for their input. For the inequation (4.2) the normal form is shown in formula (4.3).

$$x_i - x_{i+1} \leq 0 \tag{4.3}$$

For the remainder of this chapter, we will present equations in easily readable forms, given that if they are not normalized they can always be transformed into normal form.

A distance constraint k limits the distance between two tab stops, represented by their respective positions x_i and x_{i+1}. If the constraint has a minimum distance $d_{min}(k)$, the inequation (4.4) will be generated. If the constraint defines a maximum distance $d_{max}(k)$, then an additional inequation (4.5) will be generated. If $d_{min}(k)$ and $d_{max}(k)$ are the same, then the distance is fixed and only one equation (4.6) with the distance $d(k)$ needs to be created.

$$x_{i+1} - x_i \leq d_{min}(k) \tag{4.4}$$

$$x_{i+1} - x_i \geq d_{max}(k) \tag{4.5}$$

$$x_{i+1} - x_i = d(k) \tag{4.6}$$

The cell constraint is quite similar. Given a cell constraint with its set of equally sized cells $\{c_i\}$. For each cell c_i one equation (4.7) will be emitted. Depending on the direction, x_{i1} is the position of the smaller and x_{i2} the position of the larger tab stop of the cell's borders. The helper variable h represents the common width.

$$x_{i2} - x_{i1} = h \tag{4.7}$$

To specify the objective function, we use the fact that the most optimal solution will be the original solution. The basic idea is to have a penalty for deviation. The objective function has to be set up in a way so that it will be the more optimal the smaller the deviation is. One possible solution is function (4.8) (where x'_i is the original position and x_i is the new one). The target function is to be minimized with respect to the sum of all deviations of all tab stops from their original position. This deviation is the absolute value of difference of the two positions.

$$min : |x_0 - x'_0| + |x_1 - x'_1| + \ldots + |x_n - x'_n| \tag{4.8}$$

The problem is that the absolute value is a non-linear function. But we are allowed to model linear functions only. To work around this issue, we use substitution (4.10) proposed in [18], replacing the absolute values in the objective function by helper variables δ_{x_i}, turning it into the linear function (4.9).

$$min : \delta_{x_0} + \delta_{x_1} + \ldots + \delta_{x_n} \tag{4.9}$$

$$\delta_{x_i} := |x_i - x'_i| \tag{4.10}$$

However, the substitution (4.10) is also non-linear. We still need a way to encode the absolute value into the helper variable. This can be done by generating for each tab stop x_i and it's associated helper variable δ_{x_i} the inequations (4.11) and (4.12).

$$x_i - x'_i \leq \delta_{x_i} \tag{4.11}$$

$$x'_i - x_i \leq \delta_{x_i} \tag{4.12}$$

This is the intuition why it works: Since we want to optimize for the minimum, the helper variable in the target function will be pushed towards a smaller value. Anyhow, it is bounded from below by the difference between the new and the original position. No matter what direction a tab stop is moved, one of the two equations will always be greater or equal to zero, hence positive. For this reasons δ_{x_i} will always take the value of $|x_i - x'_i|$. The minimization of δ_{x_i} is hence transferred directly into a minimization of the deviation from the original.

4.2 Encoding Control Sizes

The described encoding still lacks the sizes of the cells, determined by the contents of the contained elements. These values can be retrieved using functions of the underlying frameworks. For each direction, each element (and its enclosing cell) can either be resizable or have a fixed dimension. If the size is fixed, we encode it directly by constraining the distance between the two tab stops as in equation (4.13). For a direction d of a cell c_i, the fixed dimension is given by the library function $dim_d(c_i)$, the smaller tab stop is represented by x_{i1} and the larger one by x_{i2}, respectively.

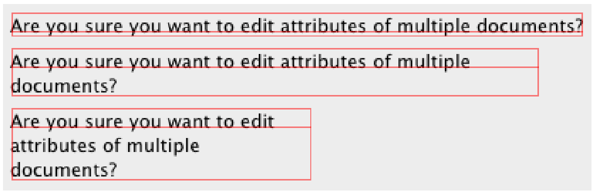
$$x_{i2} - x_{i1} = dim_d(c_i) \tag{4.13}$$

If the size is flexible, this will of course not work. A flexible cell may have a minimum dimension only, determined by a similar library function $dim_{min,d}(c_i)$. One could now create a similar inequation (4.14) for resizable cells.

$$x_{i2} - x_{i1} \geq dim_{min,d}(c_i) \tag{4.14}$$

The buttons from our example of Figure 2.1 could now be encoded using equation (4.13) for their fixed height and using inequation (4.14) for their flexible width. The buttons would always show their complete content as their minimum size is required by the specification. This way of encoding interface elements is fine for elements with one fixed dimension, such as a constant height in the case of a button. Problems arise when controls are flexible in both directions. Assuming that a resizable control has only one minimum size proves to be false.

One element exposing this behavior is the description text in Figure 4.1. Obviously, this element is resizable both in height and width. When we decrease the width, the text will wrap, creating a third line and by that increasing the minimum height. Similarly, if we make it wide enough, all text may fit on a


Are you sure you want to edit attributes of multiple documents?

Are you sure you want to edit attributes of multiple
documents?

Are you sure you want to edit
attributes of multiple
documents?

Figure 4.1: The description text from Figure 2.1 with different sizes. First wide enough to fit all text on a single line, second the original size taking two lines and third too small and hence needing three lines.

single line, thus decreasing the minimum height. This also works in the other direction, like constraining the height and computing the required minimum width. The relation between height and width can be described as a function. The graph in Figure 4.2 shows an example for the same control with contents in two different languages.

The depicted function is not linear but exposes a stepped behavior instead. When looking along the increasing x-axis, each edge represents the point when the control is wide enough to fit its contents on fewer lines. Such a step occurs depending on a wide variety of factors that are extremely implementation-dependent. Besides element properties like the font being used or the line wrapping behavior[1], this function is mainly controlled by the structure of the text, i.e., the length and the number of words. Changing the contents changes the whole function. Like it can be seen from the figure, replacing English contents with a German translation produces steps at completely different positions with different distances between them.

One could now try to simplify the problem, making it easier to handle and also making it linear such that we can use a linear solver like before. Every such simplification would deviate from the actual function (it would not be a simplification otherwise). Either the result will be too optimistic—resulting in a layout that crops contents—or it will be too pessimistic—resulting in a layout that uses more space that it actually needs to. Both options are hence not usable in our scenario. Since there are too many factors and possible corner cases, re-calculating the sizes directly in the model is not possible either. Therefore the existing function has to be used.

The exact sizing function can only be derived from the interface element itself. Hence, we need to use it directly. We can make some safe assumptions:

1. The interface element is able to tell about its minimum size. Without this assumption we would not be able to determine the sizing function at all.

2. The contents of a control will not change during the computation. If they do change, the layout would need to be re-computed from scratch anyway.

3. The sizing function is steady and is a composite of multiple linear functions. Thus, it can be described as a sorted set of data points with each adjacent pair connected by one such linear function.

[1]Maybe the control implements hyphenation, breaks words by characters or adjusts the font size dynamically.

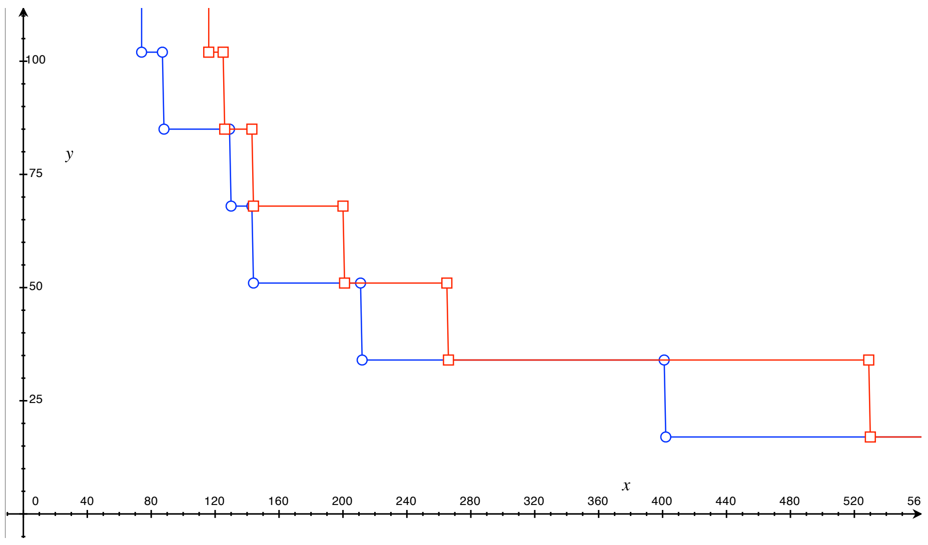


Figure 4.2: A section of the minimum size function for the control from Figure 2.1, once with the original English contents (the blue/circle line) and once with an according German translation (the red/square line). Hereby the x-axis represents the width and the y-axis the height of the element, respectively. All points above or directly on the functions are valid sizes, all others are invalid.

These assumptions allow us to determine the function in advance and encode it statically into our optimization problem. For simplicity reasons, we assume that there exists[2] a library function (4.15) $sizes(c)$ that returns these data points for (the element in) a given cell c as an ordered set of tuples.

$$sizes(c) := [(w_i, h_i)] \tag{4.15}$$

With the very precise picture of the element's minimum size function. When having a closer look at Figure 4.2, one can see that each point of the function lies at or between two adjacent data points. With each data point given as a tuple (w_i, h_i), we can express the w-coordinate of any point (w, h) between two points k and $k+1$ as a linear combination like in formula (4.16). The scalars ρ and $(1-\rho)$ represent the fractions of the points being used. The equation for the h-coordinate is very similar and can be seen in (4.17).

$$w = \rho * w_k + (1-\rho) * w_{k+1}, \qquad 0 \leq \rho \leq 1 \tag{4.16}$$

$$h = \rho * h_k + (1-\rho) * h_{k+1} \tag{4.17}$$

These formulas, however, are limited to two data points only, whereas we need a formula that expresses the whole function. This cannot be realized using the classical linear optimization described before.

lpsolve [16] implements an extension to the simplex method called *Mixed-Integer Linear Optimization* [19] that enables higher order tools like *Special*

[2]While such a function does not exist directly, one can create it from more primitive existing ones in the Cocoa frameworks of Mac OS X.

Ordered Sets [20, 21]. A special ordered set is another kind of constraint that can be formulated in an optimization problem. Each special ordered set consists of a sorted set of variables (4.18), out of which only a contiguous subset (4.19) may be non-zero (4.20). The maximum length of this subset is called the type of the special ordered set (4.21).

$$SOS_t[v_1, \ldots, v_n] \tag{4.18}$$
$$NZ := [v_a, \ldots, v_b], 1 \leq a \leq b \leq n \tag{4.19}$$
$$\forall v \in SOS_t \backslash NZ : v = 0 \tag{4.20}$$
$$|NZ| \leq t \tag{4.21}$$

Commonly used types of special ordered sets are one (SOS_1) and two (SOS_2). We will use the second one.

The functions (4.16) and (4.17) can now be re-formulated as follows: For each data point i, we use a helper variable v_i representing the fraction of it being used. In the former case all these fractions were zero, except for a fixed v_k which was ρ and a fixed v_{k+1} which was $1 - \rho$. One can observe that it is two fractions out of the set of v_is that are not 0 and that all weights sum up to exactly 1. This is exactly what the special ordered set (4.22) and the constraint (4.23) express: All weights except at most two adjacent ones must be zero, with all weights summing up to 1. The only difference between this formalization and the previous one is that the data points are no longer fixed, which is exactly what we wanted.

$$SOS_2[v_1, \ldots, v_n], \qquad v_i > 0 \tag{4.22}$$
$$v_1 + \ldots + v_n = 1 \tag{4.23}$$

At any time, these fractions must hence represent a linear combination of any two adjacent data points. All that is left to obtain the function points is to multiply these fractions with the according data point coordinates. Formulas (4.24) and (4.25) show the result.

$$w = v_1 * w_1 + \ldots + v_n * w_n \tag{4.24}$$
$$h = v_1 * h_1 + \ldots + v_n * h_n \tag{4.25}$$

Putting it all together, in order to encode the dependency between the width and the height of a cell, we use a library-provided function $sizes(c)$ (4.15) that returns an ordered set of linearly linked data points determining the sizing function. We then use a special ordered set of type two (4.22) and three linear dependencies (4.23, 4.24, 4.25) to model it into our optimization problem. The size constraints of a cell c can now be expressed as shown in formula (4.26), where the variables x_{c1}, x_{c2}, x_{c3} and x_{c4} represent the tab stop positions at the cell's left, right, lower and upper border, respectively.

$$x_{c2} - x_{c1} \geq w \qquad x_{c4} - x_{c3} \geq h \tag{4.26}$$

4.3 Achieving Optimal Results

Our algorithm does not yet produce optimal results. There exists a side-effect that results in unwanted deviations from the original layout. As shown in Figure

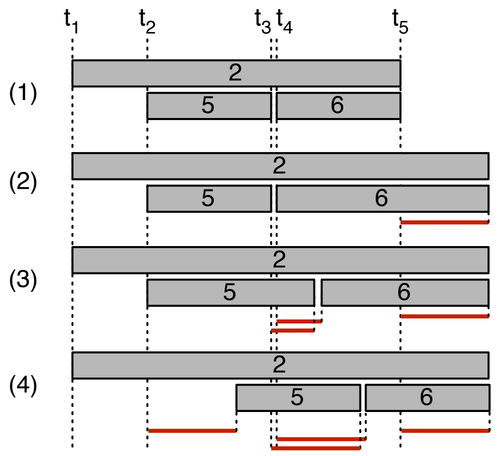


Figure 4.3: An excerpt of the cells in Figure 3.3. (1) is the original situation with the original tab stop positions and layout. In (2), the contents of Cell 2 were too large to fit the contained element, hence requiring it to increase in width. (3) is the same situation when assuming that Cells 5 and 6 are covered by a size constraint. (4) shows the actually expected result with the elements remaining in their optimal size. Shown in red are the deviations of the tab stops from their original position.

4.3, the deviation of one cell from its original size may result in a deviation of theoretically unrelated cells. Due to a content that increased in size (see situation (2)), Cell 2 has to increase its width. This happens by moving one of the border tab stops. In this case t_5 is moved to the right. Cell 6 is also attached to this tab stop and thus also increases in width. The moving of t_5 now results in an unwanted deviation from the original layout by making a cell larger that would not need to. When assuming that Cells 5 and 6 are covered by a size constraint (see situation (3)), both would increase in size, still not producing the optimal result. What we actually want is to maintain as much of the original sizes as possible (see situation (4)) which in turn means in this case to opt for a layout where the two smaller Cells retain their original size.

The reason for this behaviour can be found in the target function as we formulated it in Chapter 4.1. There, we defined a layout to be as optimal as possible, if all tab stops remain in their original position. The deviations of the tab stops from their original position (drawn in red) are to be minimized. When comparing situation (4) with both situation (2) and (3), it is visible that the latter ones are more optimal in terms of our definition. In both situations, the tab stop t_2 is not moved at all and the tab stops t_3 and t_4 are moved far less.

Obviously we have to revise the target function. The penalty of deviating from the original should now be defined in terms of equal distances. We use the tab stop graph for this definition: A layout is the more optimal the smaller the deviation of the distances of two adjacent tab stops is. Formula (4.27) shows an improved penalty function for each two adjacent tab stops t_i and t_j, with their position represented by x_i and x_j and the original position given by x'_i and x'_j,

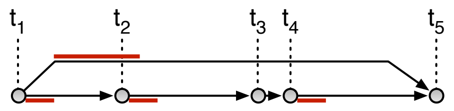


Figure 4.4: The revised penalties for the tab stop graph of the excerpt from Figure 4.3.

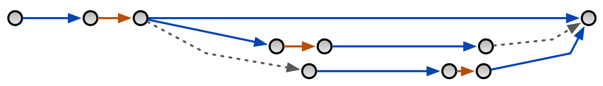


Figure 4.5: The horizontal tab stop graph from Figure 3.3 colored by types of edges. Blue are edges that go along a cell, red are edges with a distance constraint and dashed gray are edges with none of both.

respectively.

$$min: \sum_{t_i \to t_j} |(x_j - x_i) - (x'_j - x'_i)| \tag{4.27}$$

Using this target function we still do not achieve situation (4). The problem is that all edges will be increased in size simultaneously. Assuming a fixed distance between tab stops t_3 and t_4, Figure 4.4 shows the resulting penalties. The penalty (i.e., the difference in distance between the original and the new position) of tab stops t_1 and t_2 will be the same as for tab stops t_2 and t_3, as well as t_4 and t_5, respectively. The outcome will be something between situations (3) and (4) from figure 4.3—still not what we want.

To improve on this, we use the idea that some edges are not as important as others. Edges that are "used" in terms of cells and constraints are more important for the outcome than others. In our example, the edge between t_1 and t_2 is just present to ensure the ordering of tab stops. It is less important but must not be ignored completely, i.e., it must still remain in the penalty function. Having a look at Figure 4.3 and assuming that Cell 6 is gone, we still need a way to express the position of Cell 5 in the layout. Since our target function is defined in terms of length of edges only and no longer includes tab stop positions directly, every edge needs to be included.

We realize this by introducing a weight to the penalty of an edge. Figure 4.5 shows a classification of edges in the tab stop graph from Figure 3.3. According to this classification, we re-formulate our target function (4.28) and define a weighting factor σ_{ij} (4.29) for every edge in the tab stop graph, with N being the number of edges in the graph.

$$min: \sum_{t_i \to t_j} \left(\sigma_{ij} \cdot |(x_j - x_i) - (x'_j - x'_i)|\right) \tag{4.28}$$

$$\sigma_{ij} = \begin{cases} 1 & \text{if } t_i \text{ and } t_j \text{ share a cell or a dist.constr.,} \\ \frac{1}{N} & \text{otherwise.} \end{cases} \tag{4.29}$$

Every edge that is constrained by a distance constraint or that goes along a cell gets the same weight. Every other edge gets a small fraction determined by the number of edges in the graph. The target function (4.28) now yields the wanted result—like it is depicted in situation (4) of Figure 4.3.

Chapter 5

Embedding into Localization

As we mentioned previously, our layout algorithm is intended to be applied in the context of software localization. While it is generally applicable to all kinds of coordinate-based interfaces, we chose to implement it on top of Apple's Application Kit Framework [6]. We made our choice especially because of the present need for a solution of the resizing problem in this system (see Chapter 2.2). Hence, we will also embed our algorithm into the localization mechanisms of Mac OS X and our suite of tools, the Localization Suite [9], to ease the process and management of software localization.

5.1 Localization in Mac OS X

Mac OS X natively supports multiple localizations in any GUI application. This is realized by using so-called application bundles, basically a set of folders in which the actual executable resides together with all kinds of resources like icons, texts, manuals and—of course—interface files. Figure 5.1 gives an overview of how such an application bundle may look like.

These interface files contain compiled executable code that is generated from an editable file while creating the bundle. They are loaded dynamically during runtime whenever the application needs to instantiate an interface, like showing a window. The editable files are created by the developer using a visual tool. They are then compiled to executable ones and put into the application bundle.

A central element is the *strings* file format. This is a basic key-value-based file format that is used to reference localized strings. The developer references the key and the application framework will automatically find and load the according value of the variant that suits the user's preferences[1]. Here is an example of such a file:

```
"17.title" = "Cancel";
"23.tooltip" = "Close";
```

[1]On Mac OS X, each user can set up a preference list of the languages in which he wants the user interface to be displayed. In this case his preferred localization will be returned. If this is not available, his second most perfered localization will be returned and so on.

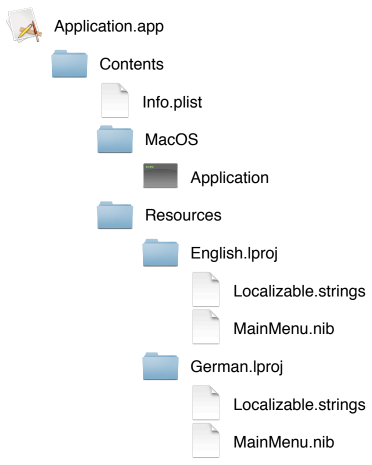


Figure 5.1: The contents of an application bundle on Mac OS X. Inside the application folder (*Application.app*) is the *Contents* folder, containing all files of the application. The *Info.plist* file holds general information about the bundle, such as the name and the version number. The *MacOS* folder contains the actual executable file *Application*. In the *Resources* folder are any kinds of resources like pictures, icons, manuals, etc. Here, we only show two language folders, *English.lproj* and *German.lproj*, respectively. These folders contain copies of all localizable resources, that may include also texts (*Localizable.strings*), interface files (*MainMenu.nib*) or other resources like pictures with translated texts in them (not shown here).

Localized variants of interface files are created using a command line tool called *ibtool* [12]. The typical process of generating a translated interface file is described in Figure 5.2. As can be seen from the figure, the first steps are to export (1) and localize (2) the strings in the interface. Step (3) creates a copy of the *original* file, replaces the original by the translated texts and writes the result as *translated* file. One could now go along and resize the translated interface manually to have all elements display their whole content. The problem is that as soon as a new (incremental) version of the *original* file exists, the *translated* file will have to be created and adjusted again.

To preserve as many of these manual adjustments as possible, *ibtool* supports the so-called *incremental localization*, depicted in Figure 5.3. Incremental localization can be more generally described as a three-way-merge [22]. From an *old original*, an old *translated* and a new *original* file, a new *translated* version is created. This process tries to accomplish two to some extent contrary goals: First, to retain all manual changes done to the *old translated* version. And second, to transfer all changes of the *original* to the *translated* version. If the changes in the new *original* and the *translated* version are disjunct, i.e., do not affect the same interface elements, these two goals can be accomplished. But as soon as there is an overlap it is no longer possible to retain both goals, weakening the first one. Instead of requiring all changes to be transferred from the *old translated* version to the new one, only as many changes as possible are retained. Changes to the original must be reflected in all localizations. Some changes in the *old translated* version however cannot preserved, as their foundation may have changed. An example is a button that was removed from the

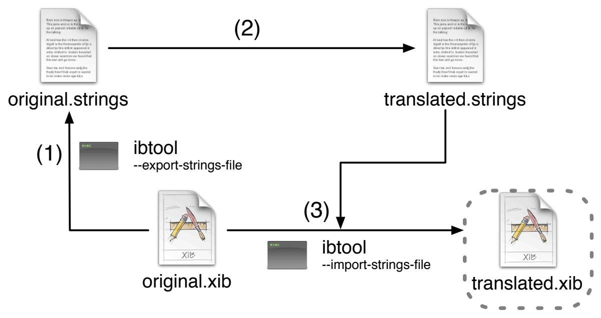


Figure 5.2: The process of generating a translated version of an interface file using *ibtool*. (1) Using the tool, all localizable strings are exported into a strings file. The result is a mapping of object ids to translatable string. (2) The file is being translated by a translator, returning a version where the keys remain unchanged but the original values have been replaced by the translated ones. (3) Using *ibtool*, the translated strings file is imported again and a copy of the original is written as the translated version.

original file. The changes to this button cannot be retained, as the basis of this change, namely the button, has gone. In order to preserve consistency, it must be removed from all localizations as well, thus making it impossible for all changes in the *translated* version to be retained.

The incremental localization can also be combined into one process with the regular one. After modifying the *original* file (Step (2), Figure 5.3) the *strings* file is created and localized (Steps (1) and (2), Figure 5.2). To create the new *translated* file, the localized *strings* are imported (Step (3), Figure 5.2) and the incremental changes are transferred (Step (3), Figure 5.3). The result is a version that contains all translations as well as possibly many of the manual changes done to the *old translated* version.

These processes, and especially the composed one, take a lot of work from the developers. Nevertheless, two major problems remain: First, the process is very complicated to be performed manually. Multiple command line calls with a couple of arguments each have to be done manually for each file. In a usual situation where a developer of a big software package has 30 languages and hundreds of interface files, this is practically impossible to perform manually. The second problem is that the interface still needs to be adjusted by hand. Solving this issue is the main topic of this work.

5.2 The Localization Suite

As software projects grow and languages are added, every developer has to cope with the effort of maintaining localizations. While the tool support given by *ibtool* is already helpful, the Localization Suite [9] is designed to automate even more aspects of localization. The suite is divided into three main tools:

- The *Localization Manager* is the hub of all tasks related to localization done by the developer. This includes the steps of importing localizable data from the files in the project, organizing communication with transla-

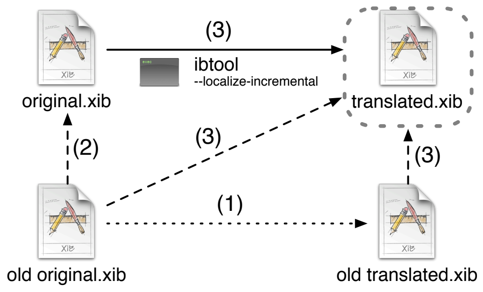


Figure 5.3: The process of transferring incremental updates from original files to a localized variant. (1) First a translated version of the original is created using the process described in Figure 5.2. (2) The developer creates a new revision of the file. (3) *ibtool* is run, creating a copy of the new original file, comparing it to the old original, and transfers the things that have not changed from the old translated file to the new one.

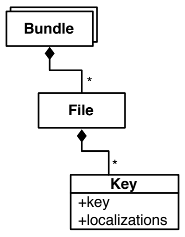


Figure 5.4: The basic data model of a localization database. It consists of a set of *Bundles*, containing a set of *Files*, again containing a set of *Keys*. A *Key* is an object that contains the actual key imported from the file, as well as the localized value for each language.

tors and creating the translated variants. The interaction with files works as described in the previous section.

- The *Localizer* is the primary tool for translators. It aids the concentration on the actual translation by a number of tools including a preview of the interface, checking for common errors or automated translation using dictionaries.

- The *Localization Dictionary* is a tool to create, maintain and deploy dictionaries, databases of localization knowledge.

The functional principle of the whole suite is to employ existing processes, using their strengths but hiding the complexity in a graphical user interface. In order to achieve this a database of all localizable data has to be maintained, as shown in Figure 5.4.

The model resembles the structure of a project, consisting of multiple *Bundles* (such as a plugin, module or component) each containing a set of localizable *Files*. Each file is converted into a list of key-value pairs during import and stored as *Key* objects. All localized variants of a key are stored inside the same object.

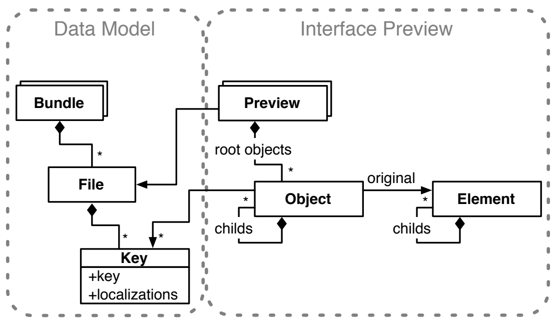


Figure 5.5: The model from Figure 5.4 extended by including loaded interfaces, called "previews". Precisely, for each file exactly one *Preview* exists. Each of these has a number of root objects (like multiple windows). These root objects form the root of an *Object* tree describing the *Element*s (i.e., controls) in the interface. Each *Object* has exactly one original, its according interface *Element*. The *Element*s also form a tree that has exactly the same structure as the *Object* tree.

To show a preview of the interface or even apply layout, we need an in-memory representation of the interface that is linked to our data model. As can be seen from Figure 5.5 two symmetric object trees, the *Object* and the *Element* tree, need to be added to the class model, both representing the same user interface. There are several reasons why this is needed:

- First of all, the *Element* tree is not as homogenous as may be guessed from the figure. Because we are using the actual elements from the underlying framework, these elements may come from hundreds of different classes. The child-relation is sometimes a to-many and sometimes to-one relationship. The name of this child property is also different between nearly all classes. In order to make this homogenous and easier to use, we use the uniform *Object* tree that mirrors the structure of the *Element* tree, but hides its diversity.
- Second, we can easily deploy the application logic. While the classes in the *Element* tree, coming from the application framework, are out of our control, the *Object* tree can be built to suit our needs. We use it to store additional metadata and transparently transfer changes to the original elements.
- We are also not able to drop the *Element* tree. While is is required to present it as a preview to the user, we also need the original classes and implementations. Without the *Element*s, we cannot determine element properties like the minimum sizing function used in Section 4.2.

5.3 Embedding the Layout Method

We now want to add our layout algorithm from Chapter 3 into this existing framework. Figure 5.6 extends the model from Figure 5.5 accordingly. The implementation of the model is pretty straightforward from the description of our

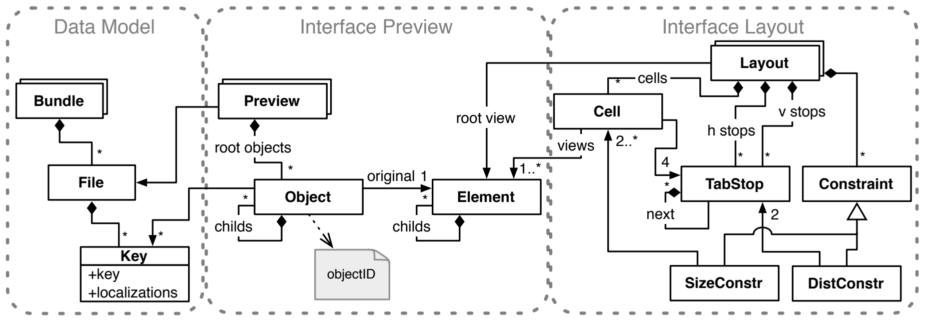


Figure 5.6: The model extended for applying the layout method using the existing framework. In addition to the model from Figure 5.5, for each root object, i.e., each window, one *Layout* can be generated using our method. Such a layout model consists of vertical and horizontal *Tab Stop*s, *Cell*s and *Constraint*s. Each *Cell* has exactly four *Tab Stop*s, one for each border, and at least one interface *Element*.

layout algorithm. Each cell consists of four tab stops and at least one element. Constraints exist in two fashions: size constraints and distance constraints. The horizontal and vertical tab stops graphs are realized using the *next* relation. The note attached to the preview objects denotes the metadata stored: for each object in the interface a unique *objectID*, that is needed for communication with *ibtool* when loading and saving changed. We can now compose the procedure of applying an automatically generated layout in the context of localization. Figure 5.7 gives an overview. The steps are in detail:

1. Previously, a *old translated* version has been generated. Using the *old model* that was retrieved from the *old original* file and an old version of the *translated strings*, an *old layouted* version was created, that might then have been modified by the developer, resulting in the *old translated* version.

2. The developer modifies the *old original* file, adding, removing or changing elements and creates a new version, the *original* file.

3. All localizable data is extracted form the *original* version, creating a *strings* file that is being sent to a translator.

4. The translator returns a file with the localization.

5. The *original* file is loaded as an interface preview like described above. Using our method from Chapter 3, a layout *model* is generated. The *translated strings* are read into the data model, triggering a re-layout of the loaded preview. This changed preview is written back to disk as the *layouted* file.

6. The same process as in step (5) is repeated, using the *old original* file and the old version of the *translated strings.*

7. By running *ibtool* the regular incremental localization is performed. Notice the difference from Figure 5.3, that instead of the *original* version, we use an already translated and *layouted* one.

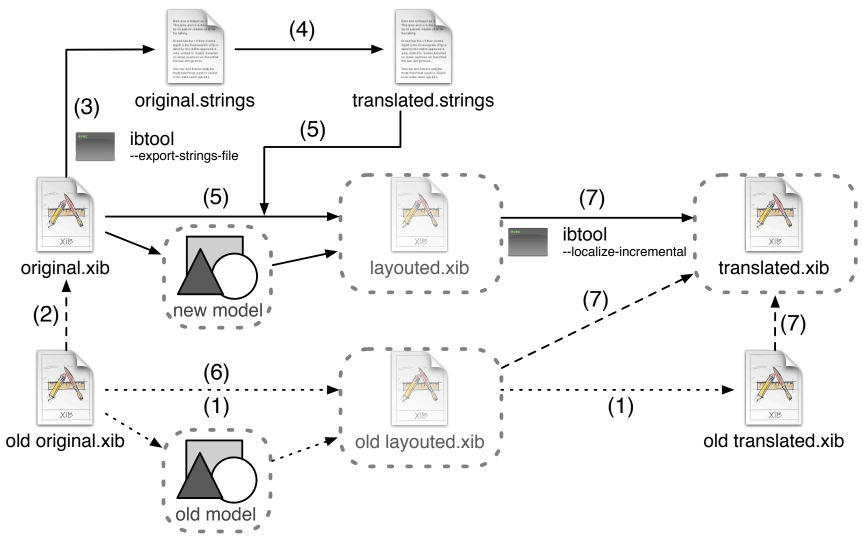


Figure 5.7: The composed localization process extended by the application of an automatically generated layout while retaining user editability of the translated file. The process is a composition of the various described elements: (1) the previous step applying the layout and creating a translated file. (2) the developer creates a new original version. (3),(4) extracting the strings and translating them. (5),(6) the generation and application of the old and new layout. (7) the incremental localization interspersed with the regular one, creating the new translated version.

This process allows the developer to have both the advantages of using automatic resizing and being able to manually fine-tune the interface. One problem still exists: if the generated model is not optimal, the results of the layout process may still need a lot of manual adjustments. For our future work, we plan to allow users to modify the automatically generated model to better suite their needs.

5.4 Discussion

One could now argue that we could have implemented a layout manager for the Application Kit. This layout manager would then take the static coordinate-based layout and resize it at runtime, also enabling the model to be used for dynamic layout as described in Chapter 2.2. Generally it should be possible to deploy our approach during runtime. Besides technical arguments, there are several reasons why we chose the way it is now:

- Our approach preserves editability. The user can open each created interface file and fine-tune the result manually. This is not possible when using a layout manager, which would have to treat every language the very same way. This is also a general drawback to be considered when using layout managers to lay out user interfaces.
- Using our approach, the resizing integrates transparently into well-known existing processes, easing the adoption by other developers.

- A custom layout manager would need to be integrated into the application, requiring modifications to the target application. Besides that, using a third-party library means loss of control and the probability of additional bugs being introduced.

Chapter 6

Related Work

Our layout model is simplified over the original ALM [10] in the features we employ. We use the central ideas of tab stops that form a graph, cells enclosing interface elements lying between these tab stops and a few very precise linear constraints. The ALM also allows arbitrary linear constraints and weak constraints with a convex penalty function for being violated. The ALM is used for both static and dynamic layout. Our method needs to generate static layouts only, with the methods of application differing widely. In our model, the controls determine their required space, whereas in the ALM the sizes are put on top of the elements using linear constraints.

The ALM has to be created manually by writing source code. Our approach, in contrast, allows the developer to create the interface using a visual tool. In order to address this shortcoming, in [23] a method similar to ours is proposed. Despite that, the targets are very different: While we want to update a static layout during localization, they use their generated model to enable dynamic layout based on static layout information only. The proposed solution is not capable of handling changing texts in interface elements, assuming that it has been well laid out previously. Their solution requires the injection of a few lines of source code into the target application whereas our solution is completely transparent. Manual modifications are still possible, but they can only happen on a layout-level. Pixel-tuning and arbitrary manual adjustments, as it is possible in our system, are not foreseen. Finally, we believe that our grouping algorithm produces better results in terms of structure recognition and independency of unrelated interface elements.

The issues of localization are a widely discussed topic amongst Mac OS developers. While Apple provides tool support for localization, the solutions feature either only very basic functionality [12] or are outdated [24]. Developers since tried to cope with the remaining issues. Basic approaches like [25] start with creating shell scripts that put some automation on top of the Apple-provided tools. These solutions however, still involve a lot of manual interaction. Tools like iLocalize [26] or the Localization Suite [9] go much further on this issue. Using separate databases and knowledge about project structures, they simplify maintaining multiple localizations. In the end, complex third-party tools are able to make up with a lot of the common problems (see Section 2.2). What they still cannot provide is an automation of the process of resizing user interfaces during localization.

[13] argues that the traditional localization process should not be used at all. In order to circumvent the problems caused by holding multiple copies of a single interface, it is suggested to use a single one which is localized at runtime. This solution compromises with nearly every aspect of localization: The interface needs to be designed in a way to accommodate all localizations, not allowing tuning for each language. Custom source code needs to be injected into the application in order to perform the run-time localization, going against the principle of transparent localization.

Few work is publicly known on solving the issue of resizing user interfaces. Among the known ones, the UILocalization classes from the Google Toolbox for Mac [27] may be the best known ones. It uses partial layout managers, which means that they affect only a few elements in an interface. Using additional views that restructure the interface, the system adapts to varying lengths of text. It uses information about struts and springs (see Chapter 2.2), by that tightly linking dynamic with static resizing which our system does not. The system does not handle height changes or interactions between multiple elements. Overall, this approach interferes with common ways of developing user interfaces, requires a lot of manual work and is limited in functionality.

While there is lots of room for improvement, we believe that our solution improves on these existing methods in several key aspects:

- Our model is more general and captures the whole interface, allowing a better informed application process and thus better results.
- It integrates transparently into existing processes and requires no source code modification of the actual target application.
- The required action of the developer has been reduced, as the system automates several steps of localization.
- Our solution allows a high grade of editability. Be it modifications of the model (which we plan to implement later on), of the resulting interface files or custom interface modifications in the target application.

Chapter 7

Conclusion

In this work, we made a proposal on how to resolve some of the issues inherently linked to coordinate-based interfaces. Precisely, we covered the issue of adjusting layout during software localization. First, we perform an automated recognition of an abstract model implied by a static interface. This abstract model is then used for propagating changes of control texts back to the coordinates. We embedded a working version of this algorithm into the Localization Suite, making it usable with only a few clicks. The main contributions of this work are the following:

1. We described a new algorithm to retrieve the underlying layout from a coordinate-based interface. While the idea of reconstructing information (also called reverse engineering) of coordinate-based interfaces is not new, the idea of grouping elements and using nested tables to create an ALM-like model is.

2. A proposal for a new method of applying this layout has been made. In contrast to existing work, we only apply changes—with the best possible case of returning the original. Though other methods use information from previously laid out interfaces and despite the fact that there are lots of layout algorithms using linear optimization, our method is new with the respect that we bind our outcome directly to the original. Also, being able to express non-linear dependencies in control sizing, we are able to have our layout method driven by the contents of controls instead of the constraints opposed on them.

3. We believe to have brought new quality to the process of localization on Mac OS X. Transparent automatic resizing is new to this platform, as far as we know. We designed our algorithm with the assumption of it never becoming perfect, no matter how far we go. Therefore it can also aid processes where it reaches its limits, i.e., produces imperfect results that can then be optimized by hand.

4. We implemented our solution and modeled it into existing tools and processes. Having it transparently integrate with existing methods bears the huge benefit of quick and easy adoption by other developers.

While our solution is integrated into Mac OS X and the Localization Suite, the concept does not depend on them. Any coordinate-based interface system should be able to perform our algorithm and probably also embed it into its own localization process.

Our system is neither finished nor perfect. The system will have to prove its effectiveness in the near future, when it will be deployed in real-world development and applications in a soon-to-be-shipped version of the Localization Suite. We can think of several future improvements:

- We should handle more corner cases during recognition, allowing more interfaces to be laid out automatically. This is especially true for nested views which are ignored in the current version.
- When cell sizes and constraints in the specification are conflicting, there is currently no way to tell which cells or constraints are at fault. This information could be helpful in finding not fitting texts.
- Manual modifications to the generated models are currently not possible. Manual changes can be done in the resulting interface files. Being able to also edit the model might make these changes superfluous.

Bibliography

[1] Alvin Yeo. Software Internationalisation and Localisation. In *Computer-Human Interaction, 1996. Proceedings., Sixth Australian Conference on*, pages 348–349, 1996.

[2] Sun. The Java Swing Layout Managers. `http://java.sun.com/docs/books/tutorial/uiswing/layout/visual.html`.

[3] Trolltech. The QT Layout Managers. `http://doc.trolltech.com/4.6/widgets-and-layouts.html`.

[4] Microsoft. Windows Presentation Foundation, part of the Microsoft .net Frameworks. `http://msdn.microsoft.com/en-us/library/ms752299(VS.100).aspx#Add_Layout`.

[5] Google. The Android User Interface. `http://developer.android.com/guide/topics/ui/declaring-layout.html`.

[6] Apple. The Application Kit Framework. `http://developer.apple.com/mac/library/navigation/index.html#Chapter=Frameworks&topic=Application%20Kit`.

[7] Apple. Interface Builder, part of the Xcode Tools. `http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/IB_UserGuide/Introduction/Introduction.html`.

[8] Apple. Localizing Nib Files. `http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/IB_UserGuide/LocalizingNibFiles/LocalizingNibFiles.html`.

[9] Max Seelemann. The Localization Suite. `http://www.loc-suite.org/`.

[10] 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, 2006. Australian Computer Society, Inc.

[11] Apple. Mac OS X. `http://www.apple.com/macosx/`.

[12] Apple ibtool. `http://developer.apple.com/Mac/library/documentation/Darwin/Reference/ManPages/man1/ibtool.1.html`.

[13] Wil Shipley. Pimp My Code, Part 17: Lost in Translations. `http://wilshipley.com/blog/2009/10/pimp-my-code-part-17-lost-in.html`, October 2009.

[14] Apple. Setting a View's Autosizing Behavior. http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/IB_UserGuide/Layout/Layout.html#//apple_ref/doc/uid/TP40005344-CH19-SW17.

[15] D. Raggett, A. Le Hors, and I. Jacobs. HTML 4.01 Specification. http://www.w3.org/TR/html401/struct/tables.html, December 1999.

[16] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. lpsolve - Open source (Mixed-Integer) Linear Programming system. http://lpsolve.sourceforge.net/5.5/, May 2004.

[17] David Gale. Linear Programming and the Simplex Method. *Notices of the AMS*, Volume 54(Number 3), March 2007.

[18] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. The lpsolve reference guide, Absolute values. http://lpsolve.sourceforge.net/5.5/absolute.htm, May 2004.

[19] M. Benichou, J. M. Gauthier, P. Girodet, G. Hentges, G. Ribiere, and O. Vincent. Experiments in mixed-integer linear programming. *Mathematical Programming*, 1(1):76–94, 12 1971.

[20] E.M.L. Beale and J.A. Tomlin. Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables. *OR*, 69:447–454, 1970.

[21] E. M. L. Beale and J. J. H. Forrest. Global optimization using special ordered sets. *Mathematical Programming*, 10(1):52–69, 12 1976.

[22] S. Balasubramaniam and Benjamin C. Pierce. What is a file synchronizer? In *MobiCom '98: Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, pages 98–108, New York, NY, USA, 1998. ACM.

[23] Christof Lutteroth. Automated Reverse engineering of hard-coded GUI layouts. In *AUIC '08: Proceedings of the ninth conference on Australasian user interface*, pages 65–73, Darlinghurst, Australia, Australia, 2008. Australian Computer Society, Inc.

[24] Apple Localization Tools. http://developer.apple.com/internationalization/localization/tools.html.

[25] Brian Dunagan. A suite of scripts for ibtool. http://www.bdunagan.com/2009/08/10/ibtool-scripts-on-google-code/, August 2009.

[26] Arizona Software. iLocalize. http://www.arizona-software.ch/ilocalize/.

[27] dmaclach. UILocalization in the Google Toolbox for Mac. http://code.google.com/p/google-toolbox-for-mac/wiki/UILocalization.