An Automated Layout Approach for Model-Driven WIMP-UI Generation

David Raneburger
Roman Popp
Institute of Computer Technology
Vienna University, Austria

Jean Vanderdonckt
Louvain School of Management
An Automated Layout Approach for Model-Driven WIMP-UI Generation

David Raneburger, Roman Popp
Institute of Computer Technology
Vienna University of Technology
Gusshausstrasse 27-29
A-1040 Vienna, Austria
{raneburger, popp}@ict.tuwien.ac.at

Jean Vanderdonckt
Université catholique de Louvain
Louvain School of Management
Place des Doyens 1
B-1348 Louvain-la-Neuve, Belgium
jean.vanderdonckt@uclouvain.be

ABSTRACT
Automated Window / Icon / Menu / Pointing Device User Interface (WIMP-UI) generation has been considered a promising technology for at least two decades. One of the major reasons why it has not become mainstream so far is that the usability of automatically generated UIs is rather low. This is mainly because non-functional requirements like layout or style issues are not considered adequately during the generation process. This paper proposes an automated layout approach that supports the explicit specification of layout parameters in device-independent and thus reusable transformation rules. Missing layout parameters are completed automatically, based on ‘Layout Hints’ under the consideration of scrolling preferences. We are aware that human intervention in the context of UI development will always be required to create high-quality UIs. Therefore, we aim to improve the generated UI by considering hints and applying heuristics, rather than solving a problem for which we believe that there is no generic solution.

Author Keywords
Automated Layout; WIMP-UI Generation; Model-driven

ACM Classification Keywords
D.2.2 Design Tool and Techniques: User Interfaces

INTRODUCTION
Millions of people worldwide use Window / Icon / Menu / Pointing Device User Interfaces (WIMP-UIs) to interact with all different kinds of devices (e.g., smartphones, tablets, ticket vending machines, etc.) on a daily basis. Model-driven UI generation offers the methodology to generate UIs for multiple devices from one high-level model, and thus reduce the development effort in comparison to traditional methods. Such high-level models are ideally device and platform independent and therefore, on a higher level of abstraction than UI models. Hence, they do not provide the means to specify details concerning the layout or style of a specific widget. These details can only be considered during the transformations, in particular when the high-level model is transformed into a concrete UI model. The additional specification of rendering details based on a device-independent model is not a very illustrative task and in most cases not even possible, because such details are device-specific. Additionally, their specification requires extra effort and they are frequently only supported to a limited extent by the transformation frameworks. Full automation, in particular inferring such details through heuristics, makes the outcome of most approaches "unpredictable" for designers. This inhibits the designer from achieving the exact look that she desires. Myers et al. state that automated UI generation usually implies that the designer compromises usability - the most determining factor of a UI - for the sake of automatic generation [24]. This is still true for current approaches and results in a low usability of automatically generated UIs [21].

Layout is one of the crucial parts for good UI usability, as the following statement from the Microsoft User Experience Guidelines underlines: "Layout is the sizing, spacing, and placement of content within a window or page. Effective layout is crucial in helping users find what they are looking for quickly, as well as making the appearance visually appealing. Effective layout can make the difference between designs that users immediately understand and those that leave users feeling puzzled and overwhelmed” [23]. Similar UI guidelines are provided by major software companies/platforms (e.g., Apple [1], Eclipse1, KDE2, etc.), to ensure a certain level of usability for their applications and consistency between them.

The research community took up the challenge to automate layout creation in the course of model-driven WIMP-UI generation about two decades ago. For example, scientist investigated layout techniques [32] and defined metrics for layout appropriateness [31]. However, the interest in the research community seemed to cease about one decade ago. Today most model-driven UI generation approaches use manually created presentation or layout models. Automatisms are only applied to a very limited extent. Why?

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. EICS12, June 2526, 2012, Copenhagen, Denmark. Copyright 2012 ACM 978-1-4503-1168-7/12/06...$10.00.

2http://techbase.kde.org/Development/Guidelines
Our approach aims to specify as much layout information as possible upfront, before the transformation, in a reusable way. The designer provides all layout information on concrete UI level of the Cameleon Reference Framework [4], which we consider the most appropriate level to specify layout information. To make these specifications reusable, they are part of the right hand side (RHS) of transformation rules. These RHSs are the UI model parts that the transformation rules create. Usually one screen of the UI is composed of the UI model parts of several transformation rules. These UI model parts are composed according to the structure of the high-level model. Such high-level models do not contain any layout specifications, thus its hierarchical structure is the only characteristic that is available for consideration for layouting. The rule that creates the container for the children does not consider their number, otherwise it would already be dependent on a certain high-level model. What we propose is to add hints for each child container that specify where it shall be placed in its parent container and to complete the layout data automatically, based on these hints. Thus, we support a seamless transition from fully automated to semi-automatic UI generation with human involvement. Our mixed-initiative approach will most probably not enable the designer to generate the UI she desires fully automatically in the first run. However, as we also think that iterative design is a crucial component of achieving high-quality user interfaces [25] and aim to reduce the number of iterations that a designer needs to develop the UI she desires in the long run. So the focus of this paper is on what makes automated UI generation an appropriate context for mixed-initiative layout.

The remainder of this paper starts with a short overview of layout mechanisms in state of the art UI generation approaches. We explain our automated layout approach in detail before we illustrate its integration in a UI generation framework. We will present the experiences gained through the implementation of our approach and discuss several issues that we consider worth for further investigation, before we draw the conclusions from our work.

**STATE OF THE ART**

The Layout of a UI can be created at design or at runtime. Compared to each other, design time approaches better support human intervention and run time approaches require less effort. This section contrasts design and runtime approaches that paved the way for our own work.

Lok and Feiner present a survey of automated layout techniques for information presentation in [18]. They state that "A presentation’s layout can have a significant impact on how well it communicates information to and obtains information from those who interact with it." Furthermore, they underline the necessity to consider high-level relationships and spatial constraints, and to provide a grammar based language to specify them. The authors admit that however powerful and expressive grammars may be, they may be difficult to use.

Vanderdonckt and Gillo summarize techniques from the area of visual design, to make them exploitable for UI design [32]. Their work gives a good overview of the multitude of options and provides layout guidelines to cope with the corresponding complexity. Further suggestions for a dynamic strategy for computer-aided visual design are provided in [3]. This work compares different placement strategies and concludes that in any case, generation may be considered as a fair starting point for manual refinement.

Designer intervention is supported by DON: the user interface presentation design assistant [16]. This approach applies design rules to generate menu and dialog box presentations, which can be customized by the designer to influence widget positioning.

Most model-driven UI development approach or frameworks that operate at design time use manually created layout or presentation models [27, 29]. Noteworthy in this context is SketchiXML [5] that provides an approach to transform hand drawn UI sketches, also known as wireframes or mock-ups, in UI specifications based on UsiXML. This approach does not work on the UI layout, but enables designer to express such a layout without much effort.

A model-based approach for layout generation at runtime is introduced by Feuerstack et al. [10]. Their layout model consists of a list of ordered statements which is interpreted at runtime. A layout model generator with a graphical editor is provided to facilitate the layout model creation for the designer. The runtime interpretation of the layout model makes their approach adaptable to new contexts of use, but it also makes the resulting UI less predictable for the designer. The same goes for the approach of Keränen and Plomp [14]. They adapt existing layouts to devices with a smaller screen size through splitting.

The Automated Interface Layout (AIL) [20] approach creates the layout of digital library query results for a given screen size at runtime. It assigns vertical space for each query result, before creating the horizontal layout for each result entry. SUPPLE, by Gajos et al. [11], is another runtime UI generation approach that automatically generates the UI layout based on a hierarchical model of widgets, having weights for selection and placement. Gajos et al. suggest that users may be willing to sacrifice predictability if the alternative has a large-enough improvement in adaptation accuracy.

Kennard and Leaney [13] define five key characteristics that any UI generation technique would need before it should
expect wide adoption and built their Metawidget approach upon them. Of particular interest in the context of layout is their fifth principle: applying multiple, and mixtures of layout. They explicitly negate the use of heuristics, as heuristics would decrease the usability of the resulting UI. Furthermore, heuristics would make the approach more unpredictable for the designer, which is especially problematic if you try to replace manually created UIs with identical, but generated ones (i.e., retrofitting). For these reasons, they exclude automated layout generation from their focus and consciously limit their approach to deriving only layout facts that are constrained by the back-end architecture. This is possible, as their Metawidget approach operates at runtime.

Runtime approaches in general have the advantage that more information is available (e.g., number of list entries). Rendering and layout decisions can be based on this knowledge. The drawback is that the outcome is less predictable for the designer. Moreover, layout creation at runtime may lead to a performance degradation. Our approach is applied at design time, to give the designer control over the look & feel of the resulting UI, without having to create an application specific layout model.

Our automated layout approach exploits the hierarchical UI structure and creates the overall screen layout bottom-up, in steps for each level. By creating the overall layout step wise, we can reduce the complexity of the problem to placing a container’s direct children. Creating more complex UIs is a matter of applying our technique with a larger number of steps, because more complex UIs have a deeper hierarchical structure. This is adequate for supporting scalability.

Let us illustrate the problem of placing a container’s direct children with a simple flight selection UI. The left side of Figure 1 depicts the widgets that are on the same structural level of this UI. In particular, they are the direct children of the container depicted at the right side of the figure. The selected level contains four buttons (Back, Next, Home and Logout) and one container (Airport Selection). This container contains the two containers Select Departure Airport and Select Destination Airport, which contain the concrete interaction widgets (e.g., drop-down lists) to select the departure and the destination airport. These widgets are not depicted and the containers are grayed out, because they are on lower levels in the UI hierarchy.

Our layout approach is built on the assumption that the size of each widget that shall be laid out is available. We will show how we satisfy this requirement in the next section. For now, the challenge is to fit the widgets displayed on the left side of Figure 1 into their container. So, how to start? We implemented our approach analogously to the real world problem of how to fit your stuff in your suitcase. Most probably you will start with the big things first, because you can fit in the smaller ones later more easily. So what does big in our context mean? We opted for three different criteria: widget area, width and height. The widgets can now be ordered according to one of these criteria and placed in their container. Overall we try to avoid scrolling. Therefore, we start placing the widgets on the upper left corner. In case that the widgets don’t fit in the container we try to rather exceed the container height than the container width, because UI guidelines [23] and other state of the art approaches [14, 20] suggest that users prefer vertical over horizontal scrolling.

After the insertion of each widget, we use a right-bottom strategy [3] to get the next placement options. Figure 2 illustrates this strategy with our running example. The Airport Selection container in Figure 1 is the widget with both
the largest width and height, and thus also the largest area. Therefore, we have to insert this widget first at insertion point 1 (i.e., position (0,0)), independently of sorting criteria. The new placement options for the next widgets according to the right-bottom strategy are insertion point 2 (position (361,0)) and insertion point 3 (position (0,211)). Insertion point 1 is already occupied and therefore removed. The next widget according to our criteria is one of the buttons. Our approach offers the designer to choose between the criteria smallest waste space and best ratio, to influence the selection of the insertion point. To find the best option, we insert the new widget at each point and subsequently cut the container at the right most and the lowest widget edge. Next, we evaluate each insertion point according to the selected criteria. Smallest waste space means that we compare the space that is left empty in the cut containers (see the scratched out areas in Figure 2). Best ratio means that we compare the width/height ratio of the cut containers with a given value.

We used the least waste space criteria to insert the back button at insertion point 3 in Figure 2 (the second best option is crossed out). After inserting each widget, we update the available insertion points. This means that the right - insertion point 4 (position (101,211)) - and the bottom - insertion point 5 (position (0,241)) - are added after the insertion of the back button. The now occupied point (insertion point 3) is not available anymore and therefore removed. Our approach repeats these steps for each widget. In case of the running example, we are able to fit all widgets in the container. The resulting UI will nevertheless not satisfy the user, because all buttons will be placed beneath the Airport Selection container and the next button will not be in the lower right part, where users might most probably expect it.

Figure 3 shows what we consider an adequate layout for our running example. Unfortunately, simply reversing the widget ordering criteria from biggest to smallest first will not lead to this UI either for two reasons. First, our criteria cannot distinguish the buttons. This makes their sequence arbitrary and most probably not the desired one. Second, the next button will be placed above the Airport Selection container instead of below. We think that figuring out the right sequence of interaction objects without the consideration of their functionality is not possible. So far, we use the ordering of the corresponding high-level model elements that they represent as a heuristic. According to our experience, this does still not lead to the right order in most cases. One reason might be that the order of elements does not matter for temporal operators that model concurrency in high-level models (e.g., the CTT [28] concurrency operator, or the Joint in discourse-based Communication Models [7]). The position of a widget on the screen however, is closely related to its functionality. To the best of our knowledge, there is no standard to specify a widget’s behavior. To solve the problem of where to place a child widget, we introduced ‘Layout Hints’.

Layout Hints
Our Layout Hints can be used to assign any UI widget to a certain region in its parent container. They support the divide and conquer principle, which means that they split a container in different regions and distribute the widgets to them. Our reasoning is that if you have less widgets to place, you have less options. This increases the chance to create the desired option, or at least an option that is as close to the desired one as without splitting.

The left side of Figure 4 shows that we distinguish three different sections - TOP, MIDDLE, BOTTOM. Each of these section is divided in three regions - LEFT, CENTER, RIGHT. Thus, we can distinguish nine different regions in total. Our Layout Hints introduce the vertical alignment options top and bottom and the horizontal alignment options left and right to identify each of these nine regions. Figure 4 depicts how these options need to be combined to target a certain region. The NO_ALIGNMENT region contains all widgets without Layout Hints. Our Layout Hints also support the specification of an integer as z-index. The z-index is used to refine the widget order of widgets within the same region. A widget with a higher z-index gets placed first. Layout Hints can be assigned to widgets either by the designer, when creating
a transformation rule, or by the framework, when it creates new widgets during the transformation process (e.g., buttons for navigation in case that containers are split). Our layout module evaluates these Layout Hints and places each widget in the specified region of its container.

The regions of a container are rarely equally populated. A common scenario is a crowded NO_ALIGMNENT region and empty LEFT or RIGHT regions. This means that the width and height of each region have to be adapted for each container. Our approach uses two steps to perform this adaptation. First, we split the container’s height between top, middle and bottom section. The width of each section is equal to the container’s width. In the second step, we split each section’s width between their left, no_alignment and right region. We use the minimum area required by each region (i.e., the sum of all its child widget areas) for this calculation. We then use our automated layout approach to place the widgets in each region. After each region, we refine the width calculation for the subsequent region(s). After we finish one section, we re-define the height calculation for the subsequent section(s). This way we create the layout for each container stepwise, through laying out each of its regions.

Table 1 shows which Layout Hints would create the layout of our "ideal" flight selection UI in Figure 3. The table shows that each region contains only one widget, which makes the problem trivial and leads to the desired layout for sure. Alternatively, the designer could use the Layout Hint alignment-y top for the home, back and logout button and ensure the correct sequence through different z-index values.

Our example illustrates that our Layout Hints enable the designer to capture non-functional requirements concerning the layout that cannot be captured in high-level models. Layout Hints only demand less effort, in comparison to a manually created layout model, if they are reusable. We therefore support their incorporation in the RHS of transformation rules. Advantages of Layout Hints with regard to the state-of-the-art are the following:

- Layout Hints are attached to the child element of a container, thus requiring no further specification in the container.
- This simply requires copy and paste of a transformation rule and the adjustment of the Layout Hint, in comparison to creating a specific layout, specified in a transformation rule, for each container.
- Layout Hints express layout specifications at the same level of abstraction than constraint-based approaches, but with less effort because of their propagation.
- Layout Hints are incremental. As opposed to algorithms and constraint-based approaches, it is not necessary to process the whole algorithm or set of constraints to be satisfied in order to obtain a layout.
- First tests suggest that our approach requires less effort than fully manual development, especially if more than one UI is developed for the same application.

The integration of our layout approach into a model-driven WIMP-UI generation framework is presented in the next section.

INTEGRATION IN A WIMP-UI GENERATION FRAMEWORK
We incorporated our layout approach in the Unified Communication Platform4 User Interface generation framework (UCP:UI) [8]. This UI generation framework uses its own EMF-based rule language to transform discourse-based Communication Models [7] into Structural UI models. We integrated our new approach in this framework in two steps. First, we extended the rule language to incorporate our Layout Hints. Second, we extended the framework’s layout module [17]. In particular, we refined its widget size calculation and adapted it to incorporate our Layout Hints.

Widget Size Calculation
We use a depth first algorithm to traverse the UI structure and create the layout bottom up. This means that only the size for non-container widgets has to be provided, as they are the leaf nodes of the UI tree. The size of containers is calculated after their layout has been created.

The most direct way to specify width and height of a widget is to set the corresponding attributes of the widget in the transformation rule. Additionally, we support the specification of width and size in Cascading Style Sheets (CSS). This allows style specifications via id, class and element. However, designers rarely use these option, because most transformation

---

**Table 1. Layout Hints for Flight Selection UI**

<table>
<thead>
<tr>
<th>Widget</th>
<th>Layout Hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home Button</td>
<td>alignment-y: top</td>
</tr>
<tr>
<td></td>
<td>alignment-x: left</td>
</tr>
<tr>
<td>Back Button</td>
<td>alignment-y: top</td>
</tr>
<tr>
<td>Logout Button</td>
<td>alignment-y: top</td>
</tr>
<tr>
<td></td>
<td>alignment-x: right</td>
</tr>
<tr>
<td>Next Button</td>
<td>alignment-y: bottom</td>
</tr>
<tr>
<td></td>
<td>alignment-x: right</td>
</tr>
</tbody>
</table>

---

4[http://ucp.ict.tuwien.ac.at](http://ucp.ict.tuwien.ac.at)

rules transform content that is only known at the time when they are applied and not at the time when they are designed. This means that a size specification at rule design time is not possible. To add flexibility in this regard, we introduced support for only width or height specifications. We then calculate the missing value automatically during the transformation process. This calculation is either based on static text, or default character numbers for widgets that handle text. Static text can be specified in the transformation rule, or is completed at the time of transformation. Default character numbers can be specified at the corresponding attributes in the domain model elements.

Width and height for a container are calculated after it has been laid out. List widgets are special containers in this regard. Their size, except for drop down lists, is determined by how many of their entries shall be visible. Our approach creates the layout for one entry and calculates the list’s size, using default values for the number of visible entries. The layout module also uses default values for each widget type’s size and width, in case that no other specification is available. All default values are defined in a configuration file, which can be adapted by the designer according to her needs.

**Layout in UCP:UI**

The integration of our layout approach in UCP:UI is illustrated in Figure 5. The layout module is part of the transformation framework, among other modules, and provides the implementation of the above presented concepts. The designer configures the layout module according to her needs.

Table 2 summarizes the configuration options for the designer. The *scrolling* options do not contain the value *none*. The reason is that this would lead to containers that have no valid layout, in case that the widgets cannot be fit in the given size. The available scrolling options guarantee that our approach is able to create a layout for any container. The *widget ordering* options influence the sequence in which widgets are inserted in their container. The designer can use the *z*-index in our Layout Hints to directly influence this widget order. Generic ordering options are hard to find, because the placement of widgets is highly dependent on their functionality and therefore application specific. The *insertion strategy* options are evaluated to sort potential insertion points for a widget. Which insertion points are available is influenced by the scrolling option. Finally, the designer can adapt the *default widget property* values for width and height of each widget type.

<table>
<thead>
<tr>
<th>Property</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>scrolling</em></td>
<td>both, horizontal, vertical</td>
</tr>
<tr>
<td><em>widget ordering</em></td>
<td>biggest/smallest first according to width, height, area</td>
</tr>
<tr>
<td><em>insertion strategy</em></td>
<td>smallest waste space, best ratio</td>
</tr>
<tr>
<td><em>default widget properties</em></td>
<td>width, height</td>
</tr>
</tbody>
</table>

**Table 2. Layout Module Configuration**

Model driven UI generation approaches usually consider device constraints, like screen size or widget toolkit, during the rendering process. This means that information like the size of the root container (i.e., frame or screen) is usually available, while the size of its sub containers is typically unknown. Designers rarely use the option to explicitly specify the size for a sub container at design time, because the exact number or type of its child widgets depends on the structure of the high-level model and is thus not available at rule design time. Our Layout Hints resolve this problem, because they are attached to the child element instead of the container. However, an exact size calculation of a container is impossible without a valid layout. So, we need the exact size of each container to calculate its layout and we need the layout to calculate its exact size. This is the good old chicken or the egg causality dilemma. We solved this problem by using the *best ratio* option in case that no size is available for a container. We obtain the size of the root container from a target device specification and calculate the size for each region based on the minute area of the direct children. The minimum area for a container is simply the sum of all its child areas and can always be calculated, as the size of the leaf nodes is always available.

Figure 5 shows that UCP:UI provides a set of basic transformation rules. These rules are independent of a certain Communication Model and guarantee that every correctly spec-
ified Communication Model can be transformed into a UI Model. The independence of a certain Communication Model is provided through a two step transformation process [12] and supports the reuse of the transformation rules. UIs that are generated with the basic rule set are fully functional and allow early testing of the application logic, but they hardly ever satisfy the user. Figure 5 shows that the designer can additionally provide more specific transformation rules. These transformation rules are added to the existing rules and usually render application domain specific concepts in the desired way. If the designer used mockups or wireframes during an initial design phase she can include the layout specifications in the UI model part (RHS) of a specific transformation rule. Direct layout specifications can be defined only within one UI model part of a transformation rule. This means that direct specification is not possible in case that one transformation rule creates the container, and other transformation rules create the UI model parts that populate it. Our Layout Hints can be used in this situation, to roughly specify the placement for a child in its container.

Let us illustrate the implications of the two-step transformation process and the effect of our Layout Hints with our running example. Figure 6 shows a transformation rule that transforms a ClosedQuestion-Answer Adjacency Pair. The upper part of the figure shows the Communication Model pattern that this rule matches (i.e., the rule’s left hand side (LHS)). The root of the LHS is the Discourse element, which contains an Adjacency Pair with an opening Communicative Act of the type ClosedQuestion and the corresponding Answer as closing Communicative Act. The content of the Closed Question is specified as many EObject. This discourse pattern models the interaction, where the system provides a list of objects and the user selects one. Our running example uses this construct to model the airport selection. The objects are Airport concepts with the two String attributes name and code.

The UI model part that is created by this transformation rule (i.e., its RHS) is depicted in the lower part of Figure 6. The root container of this rule is the ClosedQuestion-Answer panel. The attached Layout Hint assigns this panel to the top region of its parent container. Its direct children are the Heading label, the ClosedQuestion foldout list and the Send Answer button. A fold out list is a container, which in case of our transformation rule contains an abstract Output Placeholder. All Placeholders are substituted through concrete widgets during the second step of the transformation process [12]. Placeholders are quite useful, because they support the specification of generic transformation rules, independently of domain elements. On the other hand, they make the direct layout specification within one UI model part of a transformation rule tricky, because it can be replaced by more than one widget.

Figure 6 shows that the layout has been specified explicitly for each widget in the rule’s UI model part. All containers use a Grid Layout and their direct children have the corresponding Grid Layout Data attached. Each placeholder specifies an OCL statement that is evaluated on the domain model object that is transformed by the transformation rule, during the second transformation step. This step creates concrete interaction widgets according to the result of the OCL statement. For example, the OCL statement “eAllAttributes” leads to the creation of one widget for each attribute of the domain model object that is transformed. In case of our running example a label for the (Airport) name and one for the (Airport) code is created. At rule design time, however, layout data can only be specified for the placeholder widget. In case that a placeholder is replaced by more than one widget, the layout needs to be completed and adapted after the second transformation step.

In UCP: UI the transformation is completed before the resulting UI is passed on to the layout module. It is not possible for the layout module to distinguish between widgets that have been created in the first and the second step. This means that partial layout is difficult to achieve without adding extra information during the transformation. We opted for the solution to adjust the layout data directly after each second transformation step. To achieve this, we added the possibility to specify a replacement direction (either vertical or horizontal) for each placeholder widget. After the second step has been completed, we add layout data to the widgets created by this step and adjust the layout data for all widgets that need to be moved due to the new ones. We additionally added the possibility for the designer to explicitly specify whether a container shall be laid out or not after the transformation process has been completed. This ensures that the manually specified layout in the transformation rule is preserved. Furthermore we gain performance, as our approach can easily recognize whether a layout shall be created or if the size of the container can be calculated directly.
The gray part of Figure 7 shows the resulting UI after the application of our transformation rule. Our Grid Layout Data offers similar characteristics as Java Swings’s GridLayout-Data. This means that alignment and fill options can be defined in addition to the positioning (i.e., row, column, rowspan, colspan). The drop down list shows that the Output Placeholder was substituted by two labels, and that the layout was adjusted horizontally (as defined by the corresponding attribute of the Output Placeholder).

If we apply the same rule again, to transform another Adjacency Pair that models the ClosedQuestion for the Destination Airport, we would create the lower part of Figure 7. The repeated application of the same transformation rule assigns the destination airport selection again to the top region of the container. If the designer wanted to explicitly assign it to the bottom, she would have to copy the rule and adjust the Layout Hints of the container. Another problem where our Layout Hints are quite useful, are the two Send Answer buttons. Here, the user would most probably expect one button to send both answers (i.e., selected airports) at the same time. Our framework combines these buttons based on the dynamic model of the UI [30]. In effect, it creates a new button that sends all answers. This button is added to the original buttons’ closest common parent for which a layout is created. Our Layout Hints provide the possibility to assign this button to the bottom right region, which is presumably the area where a user would expect a button to submit her data or to proceed to the next screen.

Figure 3 shows the concrete UI for our running example. This UI results from applying the transformation rule in Figure 6 twice, combining the Send Answer buttons in Figure 7 to a Next button and using the Layout Hints of Table 1 in the transformation rules that create the Back, Home and Logout buttons. The Layout Hint for the Next button is automatically attached by the framework when it combines Send Answer buttons.

Our running example illustrates that the initial layout is brought to UI models in UCP:UI in three steps. First, layout data can be defined manually in the UI model part of each transformation rule. Second, the layout data is completed and adjusted by the transformation framework after the completion of each second transformation step. Third, missing layout data is calculated and added by our layout module. Figure 5 shows that the designer can finally perform additional layout modifications on the UI model. This is important, as we do not think that a specific layout can be generated fully automatically and we therefore want to provide a good entry point to compensate still existing shortcomings.

DISCUSSION & OUTLOOK

This section discusses issues that we consider worth improving and underlines decisions that we took during the development of our approach. We also highlight challenges that we see for future development and give a short outlook on our next steps.

Regarding our automated layout approach, we think that the widget ordering strategies should be extended. The challenge here is to relate the ordering to the function of each widget. One possible solution is to provide further hints or annotations (e.g., “Ordering Hints”) in the high-level model. What troubles us is that this information belongs to another level of abstraction. As a justification for this idea we can report on our experience with interaction designers in previous projects. Most of them used mock-ups that they refined iteratively in various analysis and design phases. Mock-ups can be easily created and provide the possibility to get early feedback from end users. They can then be adapted without much effort in each iteration. Additionally, they provide the advantage to capture non-functional (e.g., style or layout) requirements and explore different options. High-level task or interaction models provide the means to capture the sequence and content of the interaction, but there is already more information available if these models are created based on mock-ups. The layout information for example could be encoded in such “Ordering Hints” that are attached to the corresponding high-level model elements.

Another idea in this context is to weaken the impact of the ordering strategy. This could be achieved through a second iteration that automatically evaluates and improves the outcome of the first layout cycle. Further iterations and different evaluation criteria [3] and techniques [19] would even add more flexibility. Many evaluation approaches [26, 9] however, need to be applied at runtime, as they are based on log data which is not available at design time. Additionally, they focus more on the interaction technique than on the layout. The most important problem with automated evaluation for us is that the feedback depends on the evaluation technique and criteria [3]. Therefore, a powerful approach has to provide different placement strategies and a variety of evaluation options. It will probably take some time for the designer to figure out the right criteria and technique. In the worst case this will take even longer than the manual customizations would have taken (assuming that the result is equal in the end). We think that the probability to create a desired layout through a generic strategy is rather low. Therefore, we rather pursue the aim to automate as much as reasonable [6], and give the designer the possibility to provide layout hints or to override the automatically calculated values and to specify layout data on an appropriate level of abstraction manually.

Figure 7. Transformation Result
Human intervention requires adequate support. We claim that manipulating layout data in our graphical tree editor is more illustrative than specifying grammar based layout constraints, but we agree that a graphical "what you see is what you get" editor, probably similar to the Gummy [22] UI builder, would bring a considerable improvement. Such an editor should support the creation of transformation rules and the manual adaptations of the resulting UI model through direct manipulation of layout and style data. Another important feature would be easy support for rule duplication. This allows the designer to copy a transformation rule from the basic rule set and adapt or refine it according to her needs (e.g., the same rule with different Layout Hints).

We see one of the biggest challenges for improvement in automatically ensuring a basic amount of layout consistency between screens. We plan to exploit traces between widgets and high-level model elements to achieve this. Screen consistency will make manual customizations more complicated, because all other screens need to be checked for consistency in case of modification. We plan to take up this challenge based on the work of Egyed et al. [6] after we have provided adequate support for manual customizations. Adequate design support and automation will also be crucial to handle large scale UIs with many screens.

Internationalization is another point worth investigating. Flippable UIs have been introduce by Khaddam et al. [15] to adjust the layout to different ethnic backgrounds. We see here an additional challenge in different space requirements by the widgets due to different text lengths, which we will take up first.

We think that iterative design together with human intervention is required in any case for achieving high-quality UIs in a long term perspective. This is especially true for UIs that are large in terms of screen estate (i.e., screen size and resolution), because aesthetic issues are impossible to automate. Therefore, we plan to put our effort in providing adequate support for the designer in our next step. We plan to support direct layout manipulation together with adding/removing widgets through the implementation of the dynamic UI space management approach by Bell and Feiner [2]. Subsequently, we will refine our configuration properties based on the feedback that we gain through the application of our approach and include support for automated change propagation. The overall challenge related to human intervention will be to find the right balance between automation and manual changes with adequate guidance.

CONCLUSION

The layout of a UI is a crucial component for good usability. Fully automatic layout creation is possible. The downside is that such UIs will hardly ever be satisfying for the end user. The problem is that it is hard, if not impossible, to generate a specific UI based on heuristics. Specifying details however, is a cumbersome and labour-intensive task. Our approach uses heuristics that can be selected by the designer through configuration. Moreover, we provide the option to specify additional information in the form of Layout Hints. Layout Hints are part of a transformation rule’s UI model part. This means that they are reusable through reapplication of the transformation rule. The UI model part of such a transformation rule is on concrete UI level, which we consider the most appropriate one for layout specifications. The integration of our approach in UCP/UI proved the feasibility of our concepts and showed that it is sensible to accomplish the layout stepwise. Our approach can be used to efficiently create UIs for rapid prototyping and initial user evaluation. It is independent of a specific high-level model and can thus be incorporated in any model driven UI generation framework using transformation rules.

REFERENCES


