2015 48th Hawaii International Conference on System Sciences

Strategies for Automated GUI Tailoring for Multiple Devices

David Raneburger, Hermann Kaindl, and Roman Popp
Vienna University of Technology, Institute of Computer Technology
Vienna, Austria
{raneburger, kaindl, popp}@ict.tuwien.ac.at

Abstract

When the same graphical user interface (GUI) is being used on multiple devices with different properties, usability problems arise, especially when GUI pages are too large for a (small) screen. Scrolling is a usual approach in such a situation, but it also depends on device properties. For desk-top PCs used with a mouse, it is well-known that scrolling should be avoided. In contrast, for touch-based devices like tablet PCs or smartphones used with fingers, scrolling especially in vertical direction is widely used and accepted. So, providing several GUIs tailored for multiple devices is desirable but expensive, and it takes time.

Automated GUI generation may help, when different GUIs can be generated from the same high-level interaction design model. This is still an issue, however, since usually adaptations to high-level models have to be made manually. Our approach just requires a device specification with a few parameters for automated GUI tailoring, which employs heuristic optimization techniques. Our fully implemented approach even offers different tailoring strategies for automated GUI generation.

1 Introduction

GUIs for multiple devices need to take device characteristics like available screen space into account to avoid related usability problems. For example, on desk-top PCs used with a mouse (i.e., fine-grained pointing and no touch gestures), scrolling should be avoided [8, 10]. However, on smartphones used with fingers (i.e., course-grained pointing and touch gestures), vertical scrolling is considered part of the user experience [2, 9]. So, tailoring strategies are required, especially with regard to scrolling.

For the case that scrolling is to be avoided, we previously formulated the following *objectives* [16]:

1. maximum use of the available space,
2. minimum number of navigation clicks, and
3. minimum scrolling (except list widgets).

When taking screen space into account as a *constraint*, a GUI can be tailored for a specific device through optimizing it according to given optimization objectives (such as the ones above) and this constraint. This can even be done automatically through heuristic optimization search. Such an approach requires that alternative GUIs can be generated and evaluated, and in this way (at least in principle) compared with each other.

Our approach as presented in this paper offers three different strategies to be taken into account for automated GUI tailoring, from which only the first one was supported by our previous approach [16]:

1. tailoring without scrolling,
2. tailoring with vertical scrolling, and
3. tailoring with horizontal scrolling.

Additional usability aspects like different interaction techniques or the in-depth consideration of specific GUI guidelines are out of the scope of our approach. Such aspects are difficult to address in an automated way, so we implemented a semi-automatic approach that also allows the designer to customize generated GUIs manually [12].

The remainder of this paper is organized in the following manner. First, we present some background material, in order to make this paper self-contained. Then we explain our three tailoring strategies in more detail. After that, we explain the search space and the objective function. Based on them, we present our heuristic constraint optimization search and our automated device tailoring process, which uses the results of the optimization search. Finally, we evaluate our approach based on recent user studies and contrast it with previous approaches.

2 Background

The implementation of our automated GUI tailoring approach is based on the GUI generation framework of

1530-1605/15 $31.00 © 2015 IEEE
DOI 10.1109/HICSS.2015.67

IEEE computer society

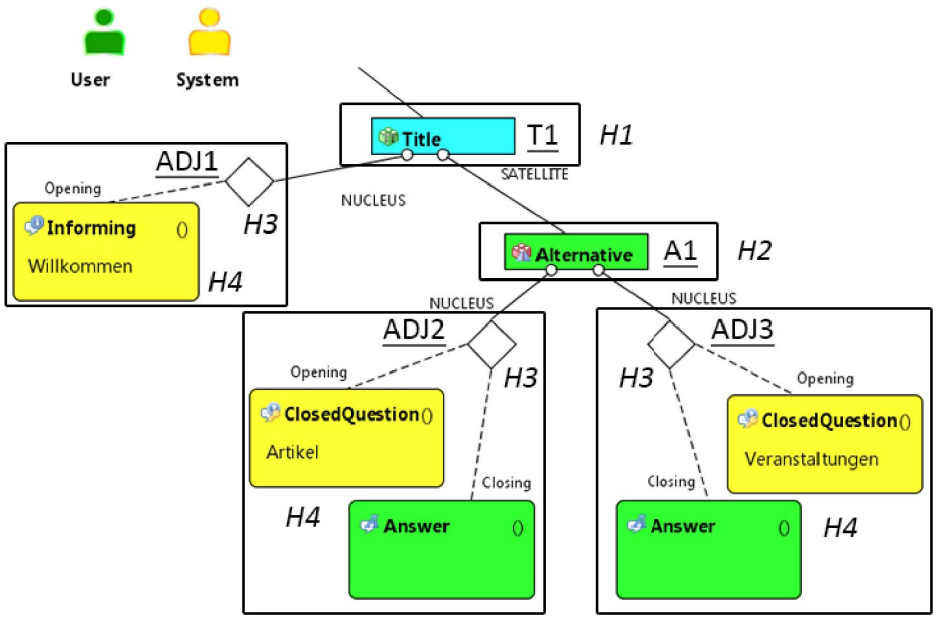

Figure 1: Vacation Planning Initial Interaction (Start Screen) Discourse Model Excerpt

the Unified Communication Platform (UCP).[1] UCP supports the use of Discourse-based Communication Models to specify the interaction between two parties device-independently [3]. Figure 1 shows an excerpt of the Discourse Model of a vacation planning application, which models the initially available interaction between the `User` and the `System` (corresponding to the start screen of the application). In particular, this model presents a welcome message and allows the user to select either an event or an article from two corresponding lists. We use this Discourse Model as a running example to illustrate our approach.

The basic interaction units of such models are `Communicative Acts`, which are depicted as rounded rectangles and assigned to one of the interacting parties, illustrated through their fill-color (green/dark for the `User` and yellow/light for the `System`). The welcome message, for example, is modeled through the `Informing` Communicative Act uttered by the `System` as shown on the left side of Figure 1. Communicative Acts are related through `Adjacency Pairs`, which are depicted as diamonds and model typically turn-takings in a conversation (e.g., question-answer or request-accept/reject). The event and the article selection are modeled through `ClosedQuestion-Answer` Adjacency Pairs. Such Adjacency Pairs can be related through `Discourse Relations`, which allow for modeling more complex flows of interaction. We use a `Title` and an `Alternative` relation in our running example, which specify that all Communicative Acts are concurrently available.

All compliant Discourse-based Communication Models can be automatically transformed to tailored GUIs using the UCP tool [11]. To achieve this, UCP provides a predefined basic set of transformation rules. In particular, it provides a *minimal basic* rule set, which supports the transformation of each compliant Communication Model to exactly one GUI. The generation of alternative GUIs, which is required to support automated tailoring, is enabled through the *additional basic* transformation rules [15].

Using these rules, UCP can generate GUIs automatically through model-driven transformations. In particular, it derives an intermediate *Screen Model* from a given Discourse-based Communication Model. For more details, see [11, 12] and below.

The properties of a given device are specified in an *Application-tailored Device Specification* [6]. It can be application-tailored in the sense that a special PC with a touchscreen may be used with fingers or mouse, respectively, which can be specified with the property *pointing granularity*. The properties specifically used by our tailoring approach are the *display resolution (x, y)* and two additional properties used by our strategies to allow for scrolling up to a certain extent (*scrollWidth* and *scrollHeight*).

3 Tailoring Strategies

We defined four strategies that provide different options for automatically tailoring a GUI for a specific device. Each

[1] `http://ucp.ict.tuwien.ac.at`

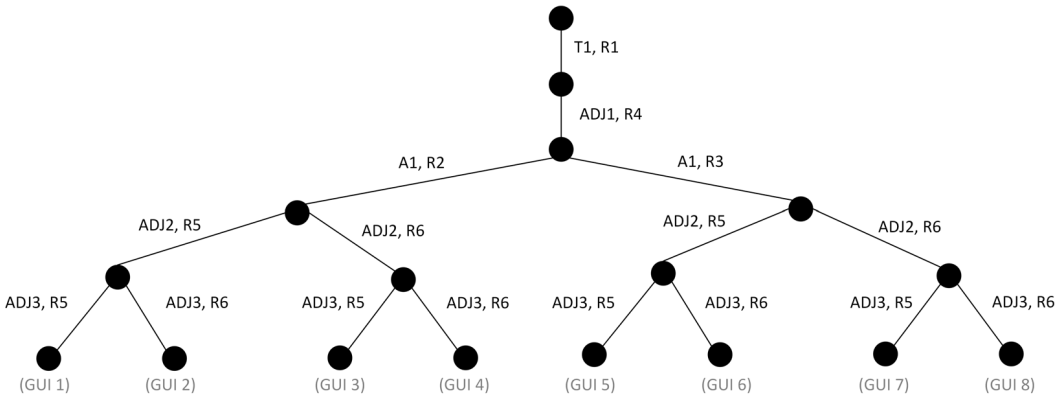

Figure 2: Search Tree for Vacation Planning Start Screen Discourse Model

of these strategies uses a different combination of properties given in the Application-tailored Device Specification. The first strategy tries to avoid scrolling, which is compliant with desktop GUI guidelines [8, 10]. The second and the third strategy make use of scrolling in either vertical or horizontal direction, which is again compliant with GUI guidelines, but this time for touch-based devices [2, 9].

1. *Screen-based Device Tailoring without Scrolling.* This tailoring strategy aims not to exceed the width and height of the device screen as specified through the properties *x-resolution* and *y-resolution*.

2. *Screen-based Device Tailoring with Vertical Scrolling.* This strategy allows for vertical scrolling, but aims not to exceed a multiple of the screen height as specified through the property *scrollHeight*. So, a virtual screen size is given by *x-resolution* and *y-resolution*$*$*scrollHeight*. This simple trick allows us to use the same implementation of our tailoring approach as for the "Screen-based Device Tailoring without Scrolling" strategy.

3. *Screen-based Device Tailoring with Horizontal Scrolling.* This strategy allows for horizontal scrolling, but aims not to exceed a multiple of the screen width as specified through the property *scrollWidth*. So, a virtual screen size is given by *x-resolution* $*$ *scrollWidth* and *y-resolution*. Again, this simple trick allows us to use the same implementation of our tailoring approach as for the first strategy.

In principle, a fourth tailoring strategy with two-dimensional scrolling is available as well. However, it is unlikely that it would generate usable GUIs.

4 Search Space

Our predefined rule set and UCP's transformation engine allow for more than one rule to be matched. This achieves the generation of more than one GUI for a given Communication Model. So, this defines a search space, where each GUI is defined through some combination of instantiated transformation rules.

Let us illustrate such a search space with our vacation planning Discourse Model shown in Figure 1. The rectangles in Figure 1 mark the five Communication Model patterns matched by the transformation rules. Table 1 lists them with their patternIDs in the first column and the matching *minimal basic* and *additional basic* transformation rules in the second and third columns. It shows that more than one transformation rule is matched for patterns A1, ADJ2 and ADJ3 each. The given pattern identifiers in addition to the rule identifiers underline that these are actually transformation rule *instantiations*.

Table 1: Transformation Rule Instantiations for Vacation Planning Start Screen Discourse Model with Hierarchy Levels

Pattern Identifier	*Minimal Basic* Rules	*Additional Basic* Rules	Hierarchy Level
T1	R1_T1	-	1
ADJ1	R4_ADJ1	-	3
A1	R2_A1	R3_A1	2
ADJ2	R5_ADJ2	R6_ADJ2	3
ADJ3	R5_ADJ3	R6_ADJ3	3

Figure 2 illustrates the search space defined by the Communication Model excerpt in Figure 1 and the predefined rule set. This search space has a tree structure. The tree's leaf nodes represent GUIs that can potentially be generated, while the other nodes on the paths represent only partly constructed GUIs. The edges are labeled with "patternID, ruleID". For example, the search tree includes only one edge for T1 and ADJ1, respectively, because each of these patterns is only matched by a single transformation rule. A1 is matched by two rules (R2 and R3), resulting in two

branches. ADJ2 and ADJ3 are also matched by two rules each, resulting in eight GUIs (GUI 1 ... GUI 8) that can be generated for our example.

5 Cost Function

In order to automatically optimize, we operationalized the given objectives into a mathematical formulation — an objective function. More precisely, we formulated it as a *cost function*, which is to be minimized.

The space actually needed by the generated screens of a GUI is key for such an optimization approach. Unfortunately, determining it exactly requires execution of the model-driven transformations and subsequent calculation of the complete layout (in a Structural Screen Model). This is computationally intensive, especially when being done for many alternative GUIs in a large search space. To reduce this computational effort, we designed our cost function in such a way that it calculates a cost value of a given GUI through evaluating the transformation rules that define it. Of course, this approach only provides estimates, but it allows indirectly comparing all GUIs that can potentially be generated before actually generating them and calculating their complete layout.

By evaluating the transformation rules corresponding to a specific GUI instead of a concrete GUI model, it is hard to achieve good estimates of the space finally required by the GUI. Fortunately, it is sufficient to provide estimates of the space requirements of the GUIs *relatively* to each other, which allow sorting the GUIs based on their corresponding transformation rule combinations. Still, these estimates have to relate to the optimization objectives.

For taking Objective 1 into account, the demand of screen space of each transformation rule's right hand side (RHS) can be estimated. The RHSs of transformation rules with the same LHS are compared, and sorted *relatively* to each other. To estimate which rule has the higher cost according to Objective 1, an estimate is determined of which RHS requires less screen space than the other ones that match the same LHS, because we want to minimize the overall cost while maximizing the use of the available space.

We defined a *relativeSpace* property for each transformation rule, which allows us to sort all transformation rules with the same LHS. This *relativeSpace* property is specified through an `integer` value and has been set to a specific value c for all *minimal basic* transformation rules, because their LHSs are mutually exclusive. When a new transformation rule is added, this property has to be set either to a higher or lower value, depending on whether the rule's RHS requires more space or less than the corresponding transformation rule from the *minimal basic* rule set. An example is the *additional basic* transformation rule that renders a single-selection item list as a drop-down box (instead of a radio button list), where both rules match `Closed-Question Answer Adjacency Pairs` (with the same defined content). We assigned this *additional basic* rule a *relativeSpace* value of $c - 1$, because its RHS requires less screen space compared to the RHS of the corresponding *minimal basic* rule.

For taking Objective 2 into account, it can be determined whether the transformation rule's RHS requires additional navigation clicks or not. This is defined as a `boolean` value of each transformation rule through a rule property named *splitting*. `True` means that the RHS splits the screen (e.g., through generating a tabbed pane) and `false` signifies that the screen is not split (e.g., through generating a panel), which requires fewer navigation clicks than the split version.

For taking Objective 3 into account, the exact space requirement would have to be determined, but see above. However, this objective is indirectly taken into account in the cost function through its operationalization of Objectives 1 and 2, because a smaller space value signifies less space consumption and more navigation clicks also result in less space consumption through splitting the screen. Both mean potentially less scrolling. Our constraint optimization approach actually tries to avoid scrolling, and this is operationalized by using the available space as a constraint, see below. Objective 3 is, however, not strict in this regard and allows scrolling, if it cannot be avoided. In this sense, it is actually operationalized as a *soft-constraint*.

Taking only the *relativeSpace* and the *splitting* properties into account for calculating the rule cost potentially results in a lot of rule combinations (corresponding to GUIs) with identical cost, although their finally required space differs and they apply splitting for different Communication Model elements. To distinguish such rule combinations, we additionally consider the type of the matched Communication Model elements (i.e., the rule's LHS) and the hierarchical position of the matched Communication Model pattern's root node in the Discourse Model tree when calculating the cost for a specific rule combination. Both are available before the GUI actually needs to be generated.

More formally, we denote such a GUI as a vector $\overrightarrow{r} = (r_1, \ldots, r_k)$, where these k transformation rule instantiations correspond to it and can (potentially) generate it (more precisely, a structural model of it).

We defined the cost of a specific combination of transformation rule instantiations ($\overrightarrow{r}$) as:

$$GUIcost(\overrightarrow{r}) = \sum_{i=1}^{k} rulecost(r_i)$$

We calculate the cost of a specific transformation rule instantiation r as:

$$rulecost(r) = -w_{relspace} * relspace(r) + w_{splitting} * splitting(r) + w_{level} * level(r)$$

The first addend $w_{relspace} * relspace(r)$ operationalizes Objective 1 and has a negative prefix, because this objective is to maximize the use of the available space, while the overall objective is to minimize the cost. $relspace(r)$ depends on the *relativeSpace* property of the transformation rule r. So, the value of $relspace(r)$ reflects the relative ordering of the transformation rules established through this property. $relspace(r)$ is equal to 0 for all *minimal basic* transformation rules where *relativeSpace* is *c*, and different from 0 otherwise.

The second addend $w_{splitting} * splitting(r)$ operationalizes Objective 2 and has a positive prefix, since this objective is to minimize the number of navigation clicks. $splitting(r)$ depends on the type of discourse element that is split (i.e., matched by the rule's LHS). We use this mechanism to reflect the semantic meaning of a specific relation through different weights. Assigning different $splitting(r)$ values for splitting `Background` or `Elaboration`, e.g., can be used to define that splitting `Background` has a lower cost than splitting `Elaboration`. Our rationale here is that the interaction specified in the `Background Satellite` does not necessarily require interaction to proceed in the discourse and will always be displayed. An `Elaboration Satellite`, in contrast, is only displayed if a specific condition is fulfilled and is typically not available initially. If this `Satellite` is displayed in an additional tab after a screen change, it will be occluded by the still available tab for the `Nucleus` and might, therefore, not be noticed by the user. $splitting(r)$ is equal to 0 for all *minimal basic* transformation rules where *splitting* is `false`, and different from 0 otherwise. i.e., the respective branches are rendered on different screens, anyway.

The third addend $w_{level} * level(r)$ is used to distinguish rule combinations where at least one transformation rule has been matched that is not part of the *minimal basic* rule set. In particular, $level(r)$ reflects the hierarchy level of the matched pattern's root node in the Discourse Model tree. For example, splitting (i.e., violating Objective 2) a `Joint` relation on a lower level has a higher cost than on a higher level. Our rationale for this choice is that elements that are related through their immediate parent are more closely related than elements via more remote ancestors (e.g., the root node) and splitting them should, therefore, have a higher cost. Similarly, creating smaller widgets (in comparison to the other transformation rules, i.e., violating Objective 1) on a lower level results in higher costs than doing so on a higher level. So, this addend is equal to 0 for all *minimal basic* transformation rules (i.e., *relativeSpace* is c and $splitting$ is `false`), and different from 0 otherwise.

So, a GUI corresponding to a combination of *minimal basic* transformation rules only, has the cost 0 by definition. This value is user defined and does not reflect the finally required space by the corresponding GUI. The cost value is only used for sorting the GUIs relatively to each other.

Each addend includes a weight ($w_{relspace}$, $w_{splitting}$ and w_{level}), which can be used to influence the impact of each addend on the overall $rulecost$. These weights can be used, e.g., to strictly minimize the use of available space before splitting a screen, or to allow for a trade-off between Objectives 1 and 2.

Let us illustrate our cost function with the Discourse Model excerpt shown in Figure 1 and the corresponding instantiations of transformation rules shown in Table 1. The last column of this table shows the hierarchy level for each transformation rule, as assigned by the cost function. The hierarchy level is also annotated in Figure 1 as *H1*, *H2*, etc. The details of how it is assigned, however, are beyond the scope of this paper.

The search space for our running example contains eight alternative GUIs (see Figure 2) defined by their corresponding transformation rule combinations. Our cost function assigns a cost value to each of these rule combinations, for ordering them. So, after applying our cost function, the following ordered list of these eight GUIs results: GUI 1, GUI 5, ..., GUI 8. This ordering reflects our optimization objectives and can be calculated entirely based on the transformation rule instantiation (i.e., without generating the corresponding GUIs).

GUI 1 is first in this ordered list and is only defined through *minimal basic* transformation rules (i.e., R1, R4 and R5). All such rules specify the $relativeSpace$ property with value c and $splitting$ as false. That is, the $rulecost$ for each such rule instantiation is equal to 0 and, therefore, also the corresponding $GUIcost$.

Our current implementation of this cost function does not separate Objectives 1 and 2 strictly, but allows for a trade-off between them with a preference to rather split a screen than to strongly reduce the use of available space. So, GUI 5 is the next GUI in the list, because it matches the *additional basic* transformation rule R3 on `Alternative A1`, which splits the screen, instead of the *minimal basic* transformation rule R2. Its cost is completely determined through the cost of R3_A1, since all other matched rules are still *minimal basic* transformation rules with $rulecost = 0$.

The GUI with the highest cost, i.e., the last GUI in our ordered list, is GUI 8, which contains all three *additional basic* transformation rules.

The factor $splitting(r)$ is calculated dynamically based on the number of elements of a certain relation, to ensure that a given "splitting order" is achieved. It is determined through a so-called "splitting-weight" assigned to each Discourse Relation that can potentially be split. For example, if splitting a `Background` relation has a lower weight

than splitting a Joint, this addend ensures that splitting all Background relations in a given Communication Model has a lower cost in total than splitting a single Joint relation. Assigning the same splitting-weights for different relations makes them indistinguishable for our cost function.

In general, it is possible that the same cost value is calculated for more than one rule combination. Each combination that has the lowest cost is optimal according to our cost function, so there may be more than one optimal solution.

6 Heuristic Constraint Optimization Search

Based on this cost function, we implemented a heuristic constraint optimization search along the lines of branch-and-bound search [7] to find an optimal GUI according to this function. In particular, we defined the available screen space given in the Application-tailored Device Specification as a constraint. When this constraint is not violated, i.e., no screen needs more space, then no scrolling is needed, either. In this case, Objective 3 is achieved, of course. Our search uses this constraint for a cut-off condition to identify potential rule combinations early that violate it. This is important as using such a branch-and-bound approach potentially allows reducing the computational effort required.

Unfortunately, as explained above, the actual space requirement of a GUI not yet generated is not available. The space component of the cost function above cannot be used as a reasonable estimate, since it only evaluates the *relative* space need. So, we defined another estimate based on further information contained in the transformation rules. A potential GUI's widgets are already defined in the RHSs of the matched transformation rules. We calculate the *minimum area* required by the RHS of a specific transformation rule through summing up the areas of all widgets (e.g., labels, buttons or text fields). The size calculation for each non-container widget is already completed through the layout module when the rule is matched (see [17]).

This *minimum area* is obviously a lower-bound of the *actually required area*, since the actual layout cannot use less space than the total size of its contained widgets. This can be formulated as:

$$minimumArea \leq actuallyRequiredArea$$

If this lower-bound on the actually required space is already larger than the available screen space, then a cut-off can already be made. The value of $availableArea$ is calculated through multiplying the values for x-resolution and y-resolution as specified. So, our cut-off condition can be formulated as:

$$minimumArea > availableArea$$

In principle, any usual search strategy (e.g., depth-first or breadth-first) may be used to traverse this search space. However, its definition allows for a special bottom-up traversal that facilitates early cut-offs. Starting with the transformation rules that match Communicative Acts or Adjacency Pairs allows identifying cut-offs early, because the *minimum area* for each node of our search space tree can be calculated directly. Starting at the root node, in contrast, requires the evaluation of all nodes in a specific branch until the first one is reached where concrete interaction widgets are specified in the corresponding transformation rule's LHS.

Let us illustrate the advantage of traversing the search space bottom-up with the Vacation Planning Discourse Model excerpt shown in Figure 1 above and the matched rules shown in Table 1 above. Our transformation rule set includes two rules (R5 and R6) that match the Adjacency Pairs ADJ2 and ADJ3. R5 creates a radio button list for the selection of a specific object (i.e., an Article for ADJ2 and an Event for ADJ3) and R6 creates a drop-down list, which requires less space. Alternative A1 in Figure 1 is matched through two rules (R2 and R3). R2 generates a panel for its sub-branches and R3 generates a tabbed-pane (i.e., the splitting property of R3 is true). That is, for R2 the size of its two child branches is summed-up, whereas for R3 only the area of the largest child branch is used. So, depending on the split property of a specific transformation rule, either the largest area of a specific child branch (i.e., splitting is true), or the sum of all child areas (i.e., splitting is false) is used as *minimum area*. Traversing the resulting search space shown in Figure 2 above bottom-up means that the transformation rules for the Discourse Model leaf nodes (i.e., Communicative Acts and Adjacency Pairs) are matched first. Such rules generate widgets required for the corresponding interaction (e.g., labels, buttons, etc.). The size for this widgets is available, i.e., the *minimum area* can be calculated immediately.

Let us assume that we want to generate a GUI for a device with a *small display* (e.g., a smartphone) and that R5 is matched first on ADJ3. Let us further assume that the *minimum area* calculated for R5's RHS fulfills our cut-off condition. So, R5 can be discarded immediately for ADJ3 and 4 search tree branches are cut off (i.e., GUI 1, GUI 3, GUI 5 and GUI 7). Next, R6 is matched on ADJ3 and the cut-off condition evaluated again. We assume here that there are no more cut-offs, so the finally resulting search space contains four branches (i.e., GUI 2, GUI 4, GUI 6 and GUI 8).

Alternatively, we want to generate a GUI for a device with a *large display* (e.g., desktop PC). Again, R5 is matched on ADJ3 first, but this time the cut-off condition is not fulfilled. Assuming that in this case only the area sum for matching R5 to ADJ2 and ADJ3 fulfills our cut-off condition, only GUI 1 in the left most branch (labeled ADJ3,R5) is discarded. So, for the large device, the finally used search space contains seven rule combinations (GUI 2 ... GUI 8).

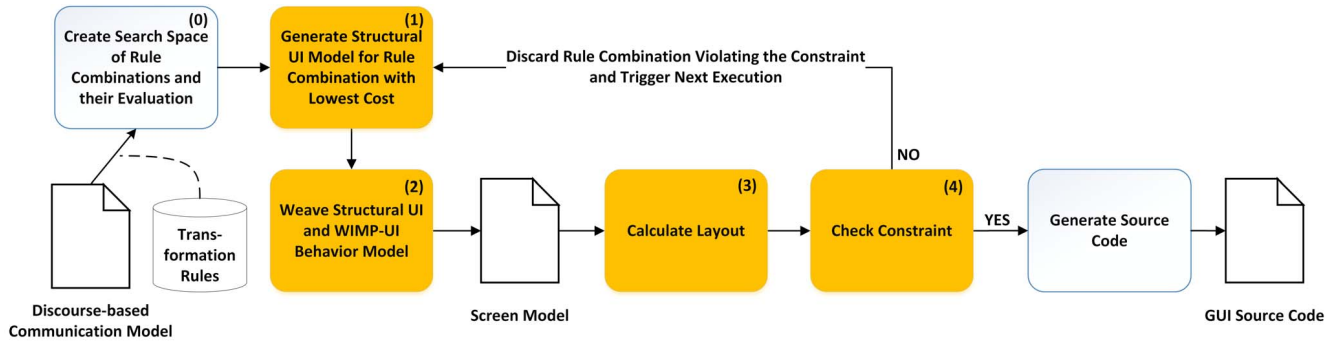

Figure 3: Automated Tailoring Process

Traversing the search space top-down would mean that transformation rule R1 on `Title T1` is matched first. This transformation rule specifies a container in its LHS for which the size cannot be estimated, because it is not defined yet which concrete interaction widgets (e.g., labels, buttons or text fields) will be contained. Next, R4 is matched on `ADJ1`, which we assume here does not fulfill our cut-off condition. Next R2 and R3 are matched on `Alternative A1`, both specifying containers in their respective LHSs. Again, the cut-off condition cannot be evaluated, because it is not defined yet which concrete interaction widgets will be contained. The cut-off condition can again be evaluated for the next nodes, where R5 and R6 are matched for `ADJ2`. From our example above, we already know that the RHSs of R5 and R6 together with the RHS of R4 do not violate our constraint, because GUI 2, GUI 4, GUI 6 and GUI 8 are contained in both resulting search spaces. So, the cut-offs, for both devices, will only be detected when R5 and R6 are matched on `ADJ3`. At this point in time, the complete search space has already been created. So, the computational effort involved would be higher, because more nodes would have to be analyzed to identify cut-offs.

Finally, the cost function is applied to all remaining rule combinations and sorted according to their costs. So, the result of the heuristic search is an ordered list of transformation rule combinations defining GUIs.

7 Automated Device Tailoring Process

Our automated process for device tailoring uses our heuristic constraint optimization search approach based on the cost function introduced above. Figure 3 shows this process with a focus on the tailoring loop, which consists of four steps (depicted as yellow rounded rectangles labeled 1 to 4).

Step 0 *"Create Search Space of Rule Combinations and their Evaluation"*, depicted on the left side of Figure 3, has to be completed only once and is a prerequisite for the automated tailoring loop. It is, therefore, not part of the loop. This step results in an ordered list of GUIs as described above. As an example, we use here the search result for the device with the larger display, which specifies seven GUIs that can potentially be generated (GUI 1 was already cut-off). So, GUI 5 corresponds to the rule combination with the lowest cost and GUI 8 to the one with the highest cost in the ordered list.

Our automated tailoring loop starts with Step 1 *"Generate Structural UI Model for Rule Combination with Lowest Cost"*, which generates the GUI for the transformation rule combination with the lowest cost. In particular, it creates the corresponding Structural UI Model, which is weaved with the WIMP-UI Behavior Model in Step 2 *"Weave Structural UI and WIMP-UI Behavior Model"*, resulting in the Screen Model [12]. Step 3 *"Calculate Layout"* calculates the layout for each screen, including the calculation of all container sizes. The execution of these three steps in each loop is required as it is not possible to determine whether a certain rule combination really violates the given constraint without knowing the exact size of each screen, which is provided by the Screen Model. Step 4 *"Check Constraint"* checks whether the size of each screen fits the space constraint of a given device.

In our running example, we generate GUI 5 in this first loop. To keep the example simple, we selected a Discourse Model excerpt that corresponds to one Presentation Unit (i.e., all Communicative Acts are concurrently available). That is, the Screen Model that is generated with *minimal basic* rules (i.e., the transformation rule combination with the lowest cost) displays all widgets concurrently and contains only one screen. This would have been the discarded GUI 1. Our tailoring process potentially splits such Presentation Units into several screens, which actually happens in GUI 5 through the use of the *basic alternative* transformation rule R3 for the `Alternative A1`. So, the Screen Model for our running example has two screens, which correspond to the same Presentation Unit. The given constraint is checked for each of these screens. Let us assume that both screens of GUI 5 satisfy it. In this case we already

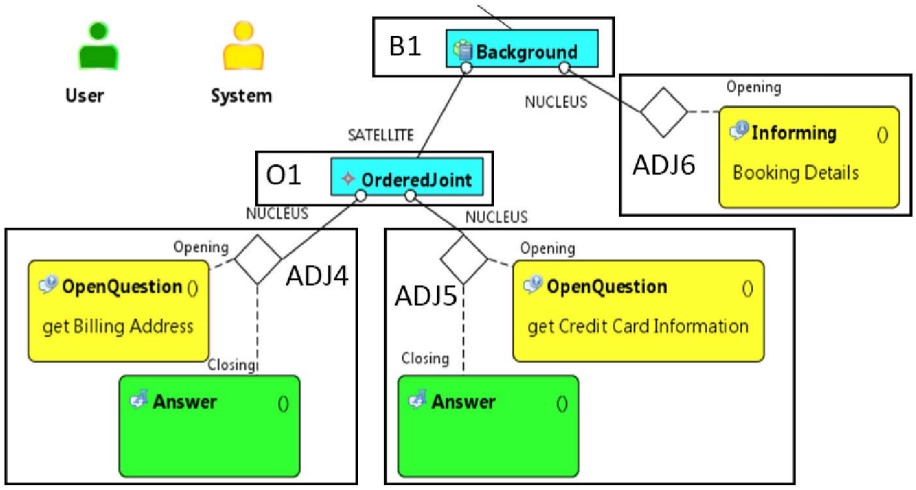

Figure 4: Vacation Planning Payment Discourse Model Excerpt

found an optimal solution according to our cost function and the source code generation is triggered, following the path labeled "YES" in Figure 3.

The tailoring process aims for a GUI without any screen that would require scrolling. So, in case that at least one of the screens does not fit, it takes the next rule combination from the sorted list, discards the current combination and triggers the *"Generate Structural UI Model for Rule Combination with Lowest Cost"* Step 1 again (following the path labeled "NO" in Figure 3). If there is no rule combination left in the sorted list, the current combination is kept as it still fulfills Objective 3, even though it requires (minimal) scrolling. So, the source code generation is triggered, following the path labeled "YES". In our running example, let us now assume that all combinations violate the given constraint. In this case, the tailoring process generates and checks all seven GUIs of the sorted list and finally delivers GUI 8 as its result. This GUI with the highest cost still fits Objective 3, because it requires the least screen space. The reason is that all possible space reductions and splits have already been made on this GUI screen.

Our screen-based tailoring approach allows for specific savings of computational effort, if an application has more than one screen, which process-oriented applications (e.g., booking applications) typically have. So, let us extend our running example with a second screen to illustrate this benefit. We use here a payment Discourse Model, in which the `System` asks the `User` for a billing address and credit card details, modeled through `OpenQuestion-Answer Adjacency Pairs` and presents a summary of the booking details, modeled through an `Informing` Communicative Act (see Figure 4). All this information is concurrently available. This is specified through an `OrderedJoint` that relates the two Adjacency Pairs and a `Background` that relates the `OrderedJoint` and the `Informing` Communicative Act. So, all Communicative Acts may be on the same screen. It could be related with the Discourse Model for the Start Screen (shown in Figure 1) through a `Sequence` relation, for example.

The Payment Discourse Model consists of five Communication Model patterns matched by the transformation rules. Table 2 shows the pattern identifier in column one, the matching *minimal basic* transformation rule instantiation in column two, and potentially available *basic alternative* transformation rules in column three.

Table 2: Transformation Rules for Vacation Planning Payment Discourse Excerpt

Pattern Identifier	*Minimal Basic* Rules	*Additional Basic* Rules
B1	R6_B1	R7_B1
O1	R8_O1	R9_O1
ADJ4	R10_ADJ4	-
ADJ5	R10_ADJ5	-
ADJ6	R4_ADJ6	-

If no rule combinations can be cut off, the complete search space with $2^5 = 32$ different combinations has to be considered. However, let us assume again that one rule combination for the start screen could be discarded through our branch-and-bound mechanism. So, the ordered list contains 31 transformation rule combinations sorted according to their costs. The rule combination with the lowest cost is the one that contains only *minimal basic* transformation rules, except for `Alternative A1`.

So, this GUI is generated first through executing Steps 1 to 3 in our tailoring process. In Step 4 the constraint is checked for each screen. First the two screens for the Start Screen are checked, which we assume here, again, satisfy

the constraint. So, an optimal solution according to our cost function has already been found for these screens and we can discard all solutions for these screens with higher cost. That is, for our running example the list is reduced to $1/8$ (i.e., four rule combinations), because eight different GUIs can theoretically be generated for the Start Screen Discourse Model. We assume here that the Payment Screen violates the given constraint. Discarding all other combinations for the Start Screen reduces the number of rule combinations with a higher cost in the ordered list to three, because the current combination can already be discarded, too. So, another optimization loop (following the path labeled "NO" in Figure 3) using the next transformation rule combination is triggered.

In this second loop, each screen is checked again. However, the implementation of our approach skips already fitting screens to reduce the computational effort. So, the tailoring process does not check the Start Screen again, but only the Payment Screens, which have been split and which we assume now to fit. As there are no more screens that violate the given constraint, an optimal solution according to our cost function has been found. So, the tailoring process triggers the source code generation (following the path labeled "YES" in Figure 3).

To further reduce the computational effort, our approach also skips screens that do *not* fit and for which no alternative rule combinations are available.

If all screens of a given Screen Model violate the given constraint, the rule combination with the highest cost is the resulting GUI, because scrolling is minimal through reduced screen space usage and additional navigation clicks.

8 Evaluation

Previous studies in the literature already exist that serve as an evaluation of the tailoring strategies presented in this paper [13], [14], and [1]. Let us summarize here briefly the evaluation approach taken and the essential results from measuring task times in two of these studies. In addition, we mention selected subjective results.

According to the user study setup (common for all three studies), each participant interacted with both HTML-based GUIs compared (depending on the respective study), recorded on video. All GUIs were generated for a simplified flight-booking application. Task times were measured and adjusted to avoid bias, e.g., through variances of times used for typing. Point-biserial Pearson correlation coefficients were calculated for these adjusted task times. Finally, subjective questionnaires were used for collecting subjective opinions.

The first and most important study [13] (with 30 participants) showed that the (adjusted) task time on a relatively small screen (typical for a smartphone) using vertical scrolling was statistically significantly smaller than using tapping (tab-based navigation, the result of the strategy that tries to avoid scrolling). In total, it took 54 percent longer to operate tapping. While 60 percent of the participants subjectively preferred vertical scrolling, only 30 percent preferred tapping.

The second study [14] (with 20 participants) showed that the (adjusted) task time on a smartphone using tapping was significantly smaller than using horizontal scrolling. On a tablet PC, in contrast, no scrolling was necessary for this latter layout. So, the (adjusted) task time on this larger device using tapping was significantly higher than for the other layout. The reason is that all information fitted on the screens, so that no scrolling was needed on this device. However, additional clicks were needed for tapping. For both devices, these results are statistically significant. The subjective preferences were fully consistent with these measured results. So, this study also provided (further) evidence that it is important to tailor GUIs to fit the size of the screen.

9 Related Work

Our previous approach to GUI optimization [16] evaluated the GUI as a whole without considering different screens. Our new *screen-based* device-tailoring provides different *tailoring strategies*, which facilitate the exploration of alternatives for different devices, in contrast to simply trying to avoid scrolling at all, while vertical scrolling is actually efficient on touch-based devices. In addition, screen-based device tailoring allows keeping an already fitting screen, and discarding all other rule combinations that apply different rules to this screen. This reduces the number of remaining rule combinations if more than one combination is applicable for the given screen. Otherwise, this screen does not have to be checked again in future automated tailoring iterations. The latter also applies for non-fitting screens. Furthermore, our screen-based approach guarantees an optimal rule combination for each screen, whereas the previous approach was not able to consider which screen was violated. Our screen-based approach still delivers an optimal solution for the remaining screens, according to our cost function, if they do not violate the constraint.

The early GADGET approach proposed optimization-based generation of the GUI layout for selecting appropriate interactors for different GUI elements [18]. GADGET used input models on widget-level and required information about the frequency of their use, in contrast to our approach, which uses high-level interaction models as input.

SUPPLE [4] introduced another approach that treats GUI generation as an optimization problem. In particular, it supported the generation of optimal GUIs for specific user abilities or devices, based on functional GUI models. These

specify what functionality should be exposed to the user. How this functionality is to be rendered was determined through different types of constraints. Compared to our high-level interaction models, such functional GUI specifications are on a lower level of abstraction and provide less flexibility for generating concrete UIs [5]. SUPPLE supported the adaptation of direct-manipulation GUIs (e.g., a text-processor GUI) for users that have not been considered as target users by the original GUI designers (e.g., motor-impaired users). Our approach, in contrast, focuses on providing GUIs for process-oriented applications (e.g., booking applications), following the gender-inclusive design principle, i.e., we aim at generating GUIs usable for all users. In fact, SUPPLE was a *user-centered* GUI generation approach, whereas the approach proposed in this paper is *usage-centered*.

10 Conclusion

The ubiquitous use of different and multiple devices entails a need for GUIs tailored to them. As compared to creating a single GUI, e.g., for a PC, providing tailored GUIs for several devices manually takes more time and is especially more costly.

While automated GUI generation from a high-level interaction model is not (yet) widely used, it has the potential to improve this situation. However, this potential may only be achieved, when the GUI generation can automatically generate different and specifically tailored GUIs from a single interaction model, without the need to adapt it manually. The new approach presented in this paper provides even different strategies for automated GUI generation, which allow specific tailoring for multiple devices through automatic optimization that takes device specifications into account.

References

[1] D. Alonso-Ríos, D. Raneburger, R. Popp, H. Kaindl, and J. Falb. A user study on tailoring GUIs for smartphones. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC'14)*, 2014.

[2] Apple Inc. *iOS Human Interface Guidelines*, August 2012.

[3] J. Falb, H. Kaindl, H. Horacek, C. Bogdan, R. Popp, and E. Arnautovic. A discourse model for interaction design based on theories of human communication. In *Extended Abstracts on Human Factors in Computing Systems (CHI '06)*, pages 754–759. ACM Press: New York, NY, 2006.

[4] K. Gajos and D. S. Weld. SUPPLE: Automatically generating user interfaces. In *Proceedings of the 9th International Conference on Intelligent User Interface (IUI '04)*, pages 93–100, New York, NY, USA, 2004. ACM Press.

[5] K. Z. Gajos, D. S. Weld, and J. O. Wobbrock. Automatically generating personalized user interfaces with supple. *Artificial Intelligence*, 174(12 - 13):910 – 950, 2010.

[6] S. Kavaldjian, D. Raneburger, J. Falb, H. Kaindl, and D. Ertl. Semi-automatic user interface generation considering pointing granularity. In *Proceedings of the 2009 IEEE International Conference on Systems, Man and Cybernetics (SMC 2009)*, San Antonio, TX, USA, Oct. 2009.

[7] A. H. Land and A. G. Doig. An automatic method for solving discrete programming problems. In M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey, editors, *50 Years of Integer Programming 1958-2008*, pages 105–132. Springer Berlin Heidelberg, 2010.

[8] Microsoft Corporation. *User Experience and Interaction Guidelines for Windows 7 and Windows Vista*, Sept. 2010.

[9] Microsoft Corporation. *Windows 8 User Experience Guidelines*, Aug. 2012.

[10] J. Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[11] R. Popp, D. Raneburger, and H. Kaindl. Tool support for automated multi-device GUI generation from discourse-based communication models. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '13)*, New York, NY, USA, 2013. ACM.

[12] D. Raneburger. Interactive model-driven generation of graphical user interfaces for multiple devices. Doctoral dissertation, Vienna University of Technology, 2014.

[13] D. Raneburger, D. Alonso-Ríos, R. Popp, H. Kaindl, and J. Falb. A user study with GUIs tailored for smartphones. In P. Kotzé, G. Marsden, G. Lindgaard, J. Wesson, and M. Winckler, editors, *Human-Computer Interaction – INTERACT 2013*, volume 8118 of *Lecture Notes in Computer Science*, pages 505–512. Springer, 2013.

[14] D. Raneburger, R. Popp, D. Alonso-Ríos, H. Kaindl, and J. Falb. A user study with GUIs tailored for smartphones and tablet PCs. In *Proceedings of the 2013 IEEE International Conference on Systems, Man and Cybernetics (SMC 2013)*, Manchester, UK, Oct. 2013.

[15] D. Raneburger, R. Popp, and H. Kaindl. Model-driven transformation for optimizing PSMs: A case study of rule design for multi-device GUI generation. In *Proceedings of the 8th International Joint Conference on Software Technologies (ICSOFT'13)*. SciTePress, July 2013.

[16] D. Raneburger, R. Popp, S. Kavaldjian, H. Kaindl, and J. Falb. Optimized GUI generation for small screens. In H. Hussmann, G. Meixner, and D. Zuehlke, editors, *Model-Driven Development of Advanced User Interfaces*, volume 340 of *Studies in Computational Intelligence*, pages 107–122. Springer Berlin / Heidelberg, 2011.

[17] D. Raneburger, R. Popp, and J. Vanderdonckt. An automated layout approach for model-driven WIMP-UI generation. In *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '12, pages 91–100, New York, NY, USA, 2012. ACM.

[18] A. Sears. Layout appropriateness: A metric for evaluating user interface widget layout. *IEEE Transactions on Software Engineering*, 19:707–719, 1993.