SUPPLE: Automatically Generating User Interfaces

Krzysztof Gajos
University of Washington
Seattle, WA, USA
kgajos@cs.washington.edu

Daniel S. Weld
University of Washington
Seattle, WA, USA
weld@cs.washington.edu

ABSTRACT

In order to give people ubiquitous access to software applications, device controllers, and Internet services, it will be necessary to automatically adapt user interfaces to the computational devices at hand (*e.g.*, cell phones, PDAs, touch panels, etc.). While previous researchers have proposed solutions to this problem, each has limitations. This paper proposes a novel solution based on treating interface adaptation as an optimization problem. When asked to render an interface on a specific device, our SUPPLE system searches for the rendition that meets the device's constraints and minimizes the estimated effort for the user's expected interface actions. We make several contributions: 1) precisely defining the interface rendition problem, 2) demonstrating how user traces can be used to customize interface rendering to particular user's usage pattern, 3) presenting an efficient interface rendering algorithm, 4) performing experiments that demonstrate the utility of our approach.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: User Interfaces; H.5.2 [**User Interfaces**]: Graphical User Interfaces (GUIs)

General Terms

Algorithms, Design, Human Factors

Keywords

adaptive user interfaces, user interface generation, optimization, constraint satisfaction, decision theory, user trace

1. INTRODUCTION

Ubiquitous computing promises seamless access to a wide range of computational tools and Internet-based services — regardless of the user's physical location. For example, when a person enters a room, room's wall display and input facilities might be allocated for that individual, allowing her to control aspects of the immediate physical environment as well as interact with the applications forming her persistent computational environment. On the other hand, if the user were in a remote location, she might use a phone, a PDA, or another mobile device to similar effect.

A critical aspect of this vision is the premise that every application (whether it be an email client or room-lighting controller) should be able to render an interface on any device at the user's disposal. Furthermore, the rendering of an interface should reflect the needs and usage pattern of individual users. Given the wide range of device types, form factors, input methods, and personal needs and interaction styles, it is unscalable for the human programmers to create interfaces for each type of device and every kind of user. Instead, an automated solution is necessary.

A number of researchers have identified this challenge and several solutions have been proposed, *e.g.* [11, 13, 14]. Although promising, the current solutions do not handle device constraints in a general enough manner to accommodate the wide range of display sizes and interaction styles available even in today's ubicomp environments. For example, the interface generators from the Pebbles project [11] make rough assumptions about the screen size (PDA) and cannot deal with situations when the most desirable rendition of an interface does not fit in the available area. iCrafter [13] relies on hand-crafted templates and XIML [14] relies on the interface designer to explicitly specify what widgets to use under what size constraints. Tools like Damask [10] greatly simplify the process of designing user interfaces for several platforms at once but even they cannot cope with the situations where device constraints cannot be anticipated in advance and where interface functionality is generated dynamically. Finally, previous systems do not address the individual needs and differences among users in rendering the interfaces.

With the SUPPLE system, we take a different approach — treating interface generation as an optimization problem. When asked to render an interface (specified functionally) on a specific device and for a specific user, SUPPLE searches for the rendition that meets the device's constraints and minimizes the estimated cost (user effort) of the person's activity. For example, Figure 2 in Section 5 depicts two different interfaces SUPPLE deemed optimal (with respect to a cost function derived from the device and user models) for two devices with the same screen size but different interaction methods; Figure 4 shows two interfaces rendered for a single device in response to different user traces. It is important to note that unlike some of the earlier work, *e.g.* [6, 12,

IUI'04, January 13–16, 2004, Madeira, Funchal, Portugal.

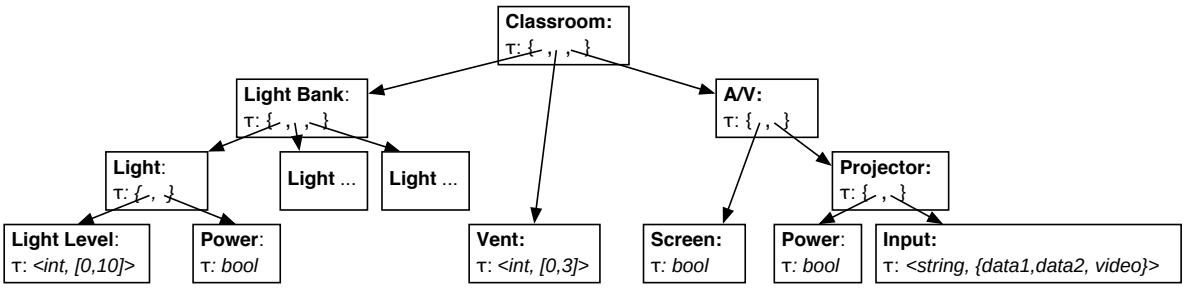

Figure 1: Tree depiction of the functional interface specification for a classroom appliance controller.

8], we not only compute the optimal layout but also choose the individual widgets to be used in rendering of the UI. In summary, this paper presents the following contributions:

- We provide a precise definition of interface generation as a constrained decision-theoretic optimization problem. This general approach allows us to generate "optimal" user interfaces given a declarative description of an interface, device characteristics, available widgets, and a user- and device-specific cost function.
- We include user traces as an essential component of the cost function. This allows SUPPLE to generate different interface renditions in response to different usage patterns.
- We present an efficient rendering algorithm which can dynamically render an interface in less than 2 seconds for all of the examples in this paper.
- We evaluate the effectiveness of our rendering algorithms with a preliminary experimental study.

2. INTERFACES, DEVICES & USERS

Intuitively, an adaptive interface renderer requires three inputs: an *interface specification* ($\mathcal{I}$), a *device model* ($\mathcal{D}$), and a *user model*, which we represent in terms of *user traces* ($\mathcal{T}$). We describe each of these components in turn.

To make this concrete, we refer to two examples throughout the paper: an interface for controlling appliances in a university classroom and an FTP client. The first example was chosen because one can plausibly imagine using very different display devices and interaction techniques for the purpose of controlling a physical environment. The FTP example was chosen because of its interactive nature.

The university classroom has several appliances that can be controlled electronically. There are three sets of lights, each with a brightness controller and an on/off switch. There is also a vent controller and an LCD projector with three possible inputs (video and two data inputs), and a mechanized screen for projection. An interface to these appliances needs to be rendered on a touch panel somewhere in the classroom. In addition, some faculty requested a capability to control the classroom directly from their PDAs and laptops.

The FTP client interface has two main views. Initially the user is asked for login information (user name, password, host name). Upon submitting this information, the user is presented with an interface that allows her to browse the local and remote file systems, upload and download files, and to choose between ASCII and binary transfer modes. The user can also choose to log out at any time and is then brought back to the login view.

2.1 Functional Interface Specification

Following earlier work on model-based UIs, *e.g.* [21, 9, 11], we adopt a functional representation — *i.e.*, one which says *what* functionality the interface should expose to the user, instead of *how* to present those features. Our rendering algorithm (Section 4) makes the latter decision.

Figure 1 illustrates the formal specification of the classroom interface. An interface is defined to be $\mathcal{I} \equiv \langle \mathcal{E}, \mathcal{C}_\mathcal{I} \rangle$, where $\mathcal{E}$ is a set of *interface elements*, and $\mathcal{C}_\mathcal{I}$ is a set of *interface constraints* specified by the designer.

The interface elements included in the description correspond to units of information that need to be conveyed via the interface between the user and the controlled appliance or application. Each element is defined in terms of its type. Formally, we define the possible types as:

$$\tau \equiv int | float | string | bool | \langle \tau, \mathcal{C}_\tau \rangle | vector(\tau) | \{{\tau_i}^{i \in 1 \ldots n}\} | \tau \mapsto nil$$

Here *int*, *float*, *string*, and *bool* are primitive types. For example, power switches in the classroom interface are defined as booleans. $\langle \tau, \mathcal{C}_\tau \rangle$ denotes a derivative type, where τ is the parent type and $\mathcal{C}_\tau$ is a set of constraints over the values of this type. In our example, light intensity and vent speed are defined as derivatives of the *int* type with added constraints on legal values (light intensity has to be between 0 and 10 and vent speed between 0 and 3). Elements of type $vector(\tau)$ denote an ordered sequence of zero or more values of tpe τ. In the specification of the FTP client, the variable holding the currently selected files is of type $vector(string)$. $\{{\tau_i}^{i \in 1 \ldots n}\}$ represents a container type, which is a sequence of elements. For example, all of the interior nodes in the specification tree in Figure 1 are instances of the container type. Finally, $\tau \mapsto nil$ denotes an action type, which returns no value of consequence to SUPPLE. There are no elements of action type in the classroom example but they are used frequently in the FTP client interface. For example, the login window is defined as an action with a container type holding the three parameters (see Figure 5(a) for a rendering).

The $\mathcal{C}_\mathcal{I}$ component of the interface description $\mathcal{I}$ represents the interface constraints, which are functions that map a full or partial rendering (defined below) to either `true` or `false`. In our example classroom interface, constraints have been added to ensure that all lights are rendered with the

same widgets. The local and remote file browsers in the FTP client are similarly constrained.

Furthermore, some additional information can be provided for each element, such as a label, an indication that an element is read only, a list of likely values or information on how to generate labels for the values (*e.g.*, even though the power switch is specified as a boolean, the individual values are represented as "On" and "Off" in a rendering in Figure 2(b)).

2.2 Device Capabilities

We model a display-based device as a tuple:

$$\mathcal{D} \equiv \langle \mathcal{W}, \mathcal{C}_\mathcal{D}, \mathcal{M}, \mathcal{N} \rangle$$

where $\mathcal{W}$ is the set of available UI widgets, $\mathcal{C}_\mathcal{D}$ denotes a set of device-specific constraints and $\mathcal{M}$ and $\mathcal{N}$ are device-specific functions for evaluating the suitability of using particular widgets in particular contexts. Below we describe all of these elements in detail.

Widgets are objects that can turn abstract UI elements into components of a rendered UI. There are two disjoint classes of widgets: $\mathcal{W} = \mathcal{W}_p \cup \mathcal{W}_c$; those in $\mathcal{W}_p$ can render primitive types, and those in $\mathcal{W}_c$ are containers, providing aggregation capabilities (i.e. layout panes, tab panes, etc.).

Like the interface constraints, the device-specific constraints in $\mathcal{C}_\mathcal{D}$ are simply functions that map a full or partial set of element-widget assignments to either `true` or `false`. For example, a constraint is used to reflect the available screen size.

We reflect certain aspects of a device's capabilities and interaction requirements by providing SUPPLE with appropriate widget libraries. For example, if the primary mode of interaction with the rendered UI is going to be touch, only widgets big enough to be manipulated with a finger may be used.

The remaining components of the device model, *i.e.* the $\mathcal{M}$ and $\mathcal{N}$ functions, reflect the estimated user effort associated with manipulating individual widgets in certain contexts.

$\mathcal{M}$ is a device-specific matching function that measures how appropriate each widget is for manipulating state variables of a given type. The effort required to manipulate some widgets may depend on the value the user wants to choose for that element. For example, the spinner widget (a text field with up and down arrows on the side, *e.g.* the light level control in Figure 4(b)) is most convenient for choosing a new value that is close to the current one. For that reason, the value of $\mathcal{M}$ may depend not only on the widget choice but also on the values it is being used for.

$\mathcal{N} : \{\mathsf{sw}, \mathsf{lv}, \mathsf{ent}\} \times \mathcal{W}_c \rightarrow \Re$ is a function, specific to container widgets, that estimates the user effort required to navigate through a rendered interface. In particular, there are three particular ways (denoted sw, lv, and ent) in which users can transition through container widgets. If we consider a widget w representing an interface element e, the three transitions are *sibling switching* (sw), when user manipulates two elements that belong to two different children of e; *leaving* (lv), when a child of e is manipulated followed by an element that is not a child of e; and *entering* (ent), which is the opposite of leaving. For different types of container widgets, these three transitions are predicted to increase user effort in different ways. For example, suppose that e represents a tab pane widget; $\mathcal{N}(\mathsf{sw}, e)$ denotes the cost of switching between its children and would be high, because this maneuver always requires clicking on a tab pane. Leaving a tab widget requires no extra interactions with the tab pane and thus incurs no cost: $\mathcal{N}(\mathsf{lv}, e) = 0$. Entering a tab pane usually requires extra effort unless the tab the user is about to access has been previously selected. In case of a pop-up window, both entering and leaving require extra effort (click required to pop-up the window on entry, another click required to dismiss it) but no extra effort is required for switching between children if they are rendered side by side.

2.3 Modeling Users with Traces

The best rendering depends on how the user will use an interface, and users with different needs may disagree on the best rendering. Thus a good estimate of the effort involved in manipulating an interface requires a usage model. Given that the goal of our research is to adapt user interfaces to users as well as devices, we chose to use *user traces* as the source of information about the intended use of the interface. A user trace, $\mathcal{T}$, is a set of *trails* where, following [20], the term trail refers to "coherent" sequences of elements manipulated by the user (the abstract elements from the interface description and not the widgets used for rendering). The important property of a trail is that if two subsequent events refer to different interface elements, this sequence of events indicates a transition made by the user. We assume that a trail ends when the interface is closed or otherwise reset. We define a trail T as a sequence of *events* u_i, each of which is a tuple $\langle e_i, v_{old_i}, v_{new_i} \rangle$. Here e_i is the interface element manipulated and v_{old_i} and v_{new_i} refer to the old and new value this element assumed (if appropriate). It is further assumed that $u_0 = \langle root, -, - \rangle$.

As the trace information accumulates, it can be used to re-render the interface adapting it to the needs of a particular user. This, of course, has to be done very carefully because frequent changes to the interface can be distracting. Also, because the format of a trace is independent of a particular rendering, the information gathered on one device can be used to create a custom rendering when the user chooses to access the application from a different device.

Of course an interface needs to be rendered even before the user has a chance to use it and generate any traces. A simple smoothing technique ensures that our algorithm works correctly with empty or sparse user traces. We note also that the designer of the interface may provide one or more "typical" user traces. In fact, if several different traces are provided, the user may be offered a choice as to what kind of usage pattern they are most likely to engage in and thus have the interface rendered in a way that best reflects their needs.

3. RENDERING AS OPTIMIZATION

Our objective is to render each interface element with a widget. Thus we define a *legal rendering* to be a mapping $\phi : \mathcal{E} \mapsto \mathcal{W}$ which satisfies the interface and device constraints in $\mathcal{C}_\mathcal{I}$ and $\mathcal{C}_\mathcal{D}$. Of course, there may be many legal renderings, and we seek the best — the one which minimizes the expected cost of user effort.[1] We use a *cost function*, **$**,

[1]We note that there are often many conflicting goals in UI design: easy to learn, fast to use for novices, suitable for experts, ease of error recovery, etc. In the future we hope

as an estimate of the user effort involved in manipulating a particular rendering of an interface. In general, we define **$** to be of the form:

$$\$(\phi, \mathcal{T}) \equiv \sum_{T \in \mathcal{T}} \sum_{i=1}^{|T|-1} N(\phi, e_{i-1}, e_i) + \mathcal{M}(\phi(e_i), v_{old_i}, v_{new_i}) \tag{1}$$

where N is an estimate of the effort of navigating between widgets corresponding to the subsequent interface elements referenced in a trail. Hence, the cost of a rendering is the sum of the costs of each user operation recorded in the trace.

In order to use Equation 1, we must first reformulate it so that N is defined in terms of the primitive navigation function, $\mathcal{N}$, which is part of the device model. Furthermore, equation 1 requires re-analyzing the entire user trace each time a new cost estimate is necessary, so we wish to transform the cost function into an algebraic form that allows incremental computation on an element-by-element basis.

Recall that our interface specification is a hierarchy of interface elements. If we assume a rendition where no shortcuts are inserted between sibling branches in the tree describing the interface, we can unambiguously determine the path between any two elements in the interface.[2] We denote path between elements e_i and e_j to be $p(e_i, e_j) \equiv \langle e_i, e_{k_1}, e_{k_2}, \ldots, e_{k_n}, e_j \rangle$. We can thus choose the navigation cost function, N, from Equation 1 to be of the form:

$$N(\phi, e_{i-1}, e_i) = \sum_{k=1}^{|p(e_{i-1}, e_i)|-2} \begin{cases} \mathcal{N}(\mathsf{sw}, \phi(e_k)) & \text{if } \mathrm{child}(e_k, e_{k-1}) \\ & \quad \wedge \mathrm{child}(e_k, e_{k+1}) \\ \mathcal{N}(\mathsf{lv}, \phi(e_k)) & \text{if } \mathrm{child}(e_k, e_{k-1}) \\ & \quad \wedge \mathrm{child}(e_{k+1}, e_k) \\ \mathcal{N}(\mathsf{ent}, \phi(e_k)) & \text{if } \mathrm{child}(e_{k-1}, e_k) \end{cases} \tag{2}$$

In the above formula, we iterate over the intermediate elements in the path, distinguishing among the three kinds of transitions described in the previous section. If both e_{k-1} and e_{k+1} are children of e_k, then we consider this to be a sibling *switch* between the children of e_k. If e_{k-1} is a grand child of e_{k+1} then the path is moving up the interface description hierarchy and we say that we *leave* e_k. Finally, if the path is moving down the hierarchy, we are *entering* e_k.

The cost of navigation thus defined, it is easy to see that the total navigation-related part of the cost function is dependent on how many times individual interface elements are found to be on the path in the interactions recorded in the user trace. We thus define appropriate count functions: $\#_{\mathsf{sw}}(\mathcal{T}, e)$, $\#_{\mathsf{ent}}(\mathcal{T}, e)$ and $\#_{\mathsf{lv}}(\mathcal{T}, e)$. Smoothing towards the uniform distribution (by adding a constant to each count) ensures that we avoid the pathological situations where some of the weights are 0.

In our current implementation we have further chosen to use a match function, M, that is independent of the particular values assigned to the widgets by the user during the interaction. Therefore, our match function depends only on the choice of a widget for a given interface element, and we may state the component cost of an interface *element*, $\phi(e)$, as:

$$\begin{aligned} \$(\phi(e), \mathcal{T}) = \quad & \#_{\mathsf{sw}}(\mathcal{T}, e) \times \mathcal{N}(\mathsf{sw}, \phi(e)) \\ + \quad & \#_{\mathsf{ent}}(\mathcal{T}, e) \times \mathcal{N}(\mathsf{ent}, \phi(e)) \\ + \quad & \#_{\mathsf{lv}}(\mathcal{T}, e) \times \mathcal{N}(\mathsf{lv}, \phi(e)) \\ + \quad & \#(\mathcal{T}, e) \times M(\phi(e)) \end{aligned} \tag{3}$$

The total cost of the rendering can be thus reformulated in terms of the component elements as

$$\$(\phi, \mathcal{T}) = \sum_{e \in \mathcal{E}} \$(\phi(e), \mathcal{T}) \tag{4}$$

This cost can now be computed incrementally, element-by-element, as the rendering is constructed. As a simple optimization, the values of the count functions are precalculated, once per interface/trace combination.

We are now ready to define our problem formally. Specifically, we define an *interface rendering problem* as a tuple $\langle \mathcal{I}, \mathcal{D}, \mathcal{T} \rangle$, where $\mathcal{I}$ is a description of the interface, $\mathcal{D}$ is a device model specifying the size constraints and available widgets and $\mathcal{T}$ is the user trace. ϕ is a *solution* to a rendering problem if ϕ is a legal rendering with minimum cost.

```
optimumSearch(vars, constraints)
1.  if propagate(vars, constraints) = fail) return
2.  if currentCost(vars)
      +remainingCostEstimate(vars) ≥ bestCost
        return
3.  if completeAssignment(vars)
4.      bestCost ← cost
5.      bestRendition ← vars
6.      return
7.  var ← selectUnassignedVariable(vars)
8.  for each value in orderValues(getValues(var))
9.      setValue(var, value)
10.     optimumSearch(vars, constraints)
11. restoreDomain(var)
12. undoPropagate(vars)
13. return
```

Table 1: Branch and bound optimization applied to the rendering problem. The variables (*vars*) represent widget-to-element assignments and are passed by reference. The *constraints* variable contains both the interface and device constraints.

4. RENDERING ALGORITHM

Table 1 shows our algorithm for solving the interface-rendition problem. The algorithm, a branch-and-bound constrained search, is guaranteed to find an optimal solution. The *remainingCostEstimate* function referenced in line two of the *optimumSearch* procedure is an admissible heuristic for pruning partial solutions whose cost is guaranteed to exceed that of the best solution found so far. This function looks at all unassigned variables and adds the costs of the best available widgets for each of these variables (ignoring constraints).

The search is directed by the *selectUnassignedVariable* subroutine. Since all variables are eventually considered,

to consider alternative cost models to see if our approach extends to these cases.

[2]Relaxing the assumption of tree structure is a key element of our ongoing research.

the order in which they are processed does not affect completeness, but (as researchers in constraint satisfaction have demonstrated) the order can have a significant impact on solution time. We have experimented with three variable ordering heuristics: *bottom-up* chooses the variable corresponding to the deepest widget in the interface specification tree (Figure 1), which leads to construction of the interface starting with the most basic elements, which then get arranged into more and more complex structures. *Top-down* chooses the top-most unassigned variable; this causes the algorithm to first decide on the general layout and only then populate it with basic widgets. Our final heuristic, *minimum remaining values* (MRV), has proven highly effective in other constraint satisfaction problems [15]; the idea is always to focus on the most constrained variable, *i.e.*, the one with the fewest number of possible values remaining.

We have further optimized the algorithm by implementing full constraint propagation at each step of the search. The constraint propagation ensures that after each variable assignment, the potential widgets considered for unassigned variables are consistent with all constraints. This allows us to more quickly detect paths that will not yield a legal solution. For example, suppose a (large) slider widget was chosen for the vent speed; constraint propagation might immediately realize that there was no way to fit a widget for light intensity. Furthermore, it allows us to make more accurate estimates of the final cost of the complete interface allowing for more efficient branch-and-bound pruning. As shown in Section 5.1, this optimization proved very effective.

Before arriving at this algorithm, we also implemented one based on simulated annealing. That algorithm was very fast at producing a legal solution, but did not guarantee optimality. Furthermore, once we optimized the systematic algorithm, local search lost its speed advantage as well.

Our algorithm is inherently discrete, while some aspects of a UI are better modeled with continuous values. For example, the length of a list widget for listing local and remote files in the FTP Client interface (Figure 5(b)) can vary reasonably from 4 to 40 entries. Our system addresses the problem by providing a few discretized approximations (e.g. lists of 5, 10, 15 elements). An alternative would be to use an optimization method capable of handling continuous parameters, but this complexity does not seem worthwhile.

5. EVALUATION

This section presents our preliminary evaluation of SUPPLE. We first report on computational efficiency. Next, we show some of the interface renderings generated automatically for different devices and for different usage patterns; these include a multi-window interface generated for the FTP client application. Finally, we describe an experiment in which we asked human experts to design the user interfaces under the same constraints as those imposed on our system.

5.1 Performance

Table 2 summarizes the results of our performance study. With no constraint propagation, the algorithm took up to 13 seconds to find the best solution and nearly 6 minutes to complete. Enabling full constraint propagation brought the numbers down to less than a second for the best solution and less than 10 seconds to complete. Varying variable ordering produced mixed results, however MRV proved to behave most predictably always completing search within 2 seconds, even though both top-down and bottom-up outperformed it on device 1.

propa-gation	var ordering	Device 1 best	Device 1 complete	Device 2 best	Device 2 complete	Device 3 best	Device 3 complete
none	bottom-up	9.40	89.10	13.04	14.45	3.15	351.29
FC	bottom-up	0.49	0.91	2.77	2.88	2.20	70.83
full	bottom-up	0.18	0.23	0.81	0.84	0.96	9.54
full	MRV	1.91	1.91	0.61	0.67	0.76	1.28
full	top-down	0.17	0.32	1.39	1.44	12.89	14.95

Table 2: Performance results for the classroom interface rendered on three different touch-based devices chosen to be particularly challenging (neither under- nor over-constraining the search space). There were over 1.8×10^9 potential renderings to explore. The numeric values represent time in seconds. The first two columns list the optimizations used: three options for propagation (none, forward checking (FC) or full) and three different variable orderings (bottom-up, minimum remaining values (MRV), and top-down) – all described in Section 4. The first column for each device shows the time necessary to arrive at the best rendering. The second column reports the time taken till the search completed (confirming that there was no better rendering to be found). The results were collected on a P4 1.5GHz computer with 512MB RAM and running Red Hat Linux 7.3 with Sun Java SDK 1.4.2.

5.2 Adapting To Device Characteristics

Figures 2, 3 and 4 show several different renditions of the classroom interface. Figures 2(a) and 2(b) show the interface rendered for two different devices with the same display size. On the first device, the interaction is pointer based. The second uses a touch screen. Figure 4(a) shows the same interface rendered on a device with a smaller screen size. In order to fit the interface in the available screen space, a less convenient widget was used for choosing the input to the projector and the light controllers were placed in individual tabs in a tab pane.

Finally, Figure 3 shows the classroom interface rendered on a cell phone. The top page, in addition to the widgets for controlling the A/V equipment and the vent, also contains a link to another page containing the controls for the lights. The figure shows the steps necessary to manipulate the brightness of one of the lights.

In order to produce these different renderings of the classroom interface, we only varied the device model. In all cases we used the same functional interface specification and an empty user trace.

5.3 Adapting To Usage Patterns

Figure 4 shows the classroom interface rendered for a smaller device. Both renditions were obtained under the same conditions with the only difference being the user trace. The rendition in Figure 4(a) was based on an empty trace and the one in Figure 4(b) was generated after heavy use of the light controls was recorded. The second rendition, even though it uses less convenient widgets and only a portion of the available space, makes it easy to navigate between individual light controls.

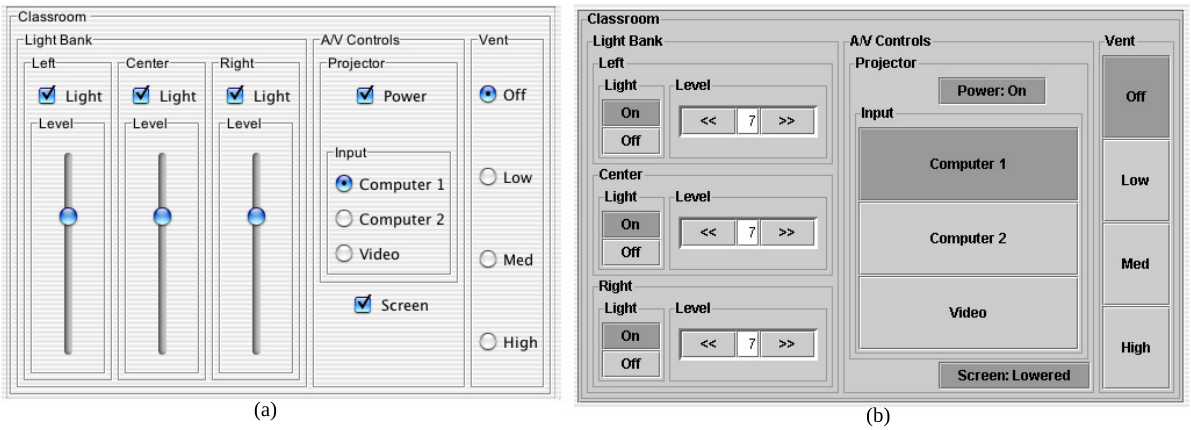

Figure 2: The classroom interface rendered for two devices with the same size: (a) a pointer-based device (b) a touch-panel device

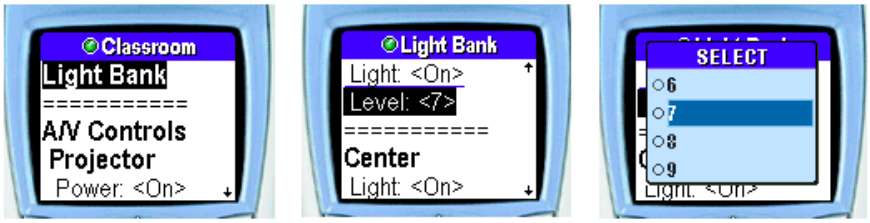

Figure 3: The classroom interface rendered on a WAP cell phone simulator (Sony Ericsson T68i); the successive screen shots illustrate the steps necessary to manipulate the brightness level of one of the lights.

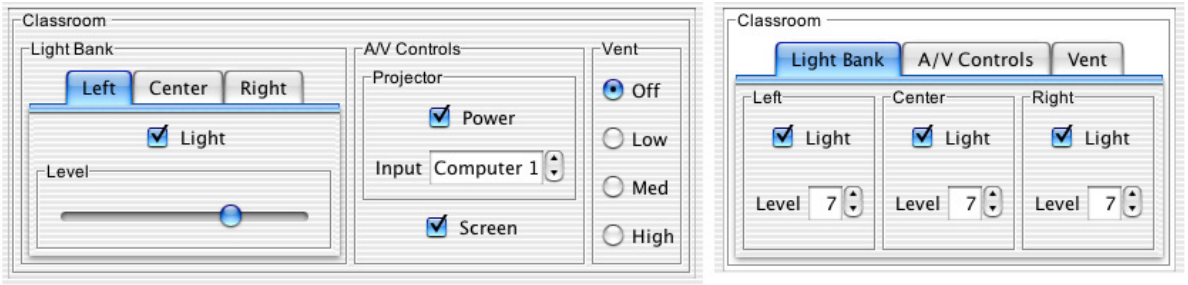

Figure 4: The classroom interface rendered on another, smaller, device: (a) with an empty user trace (b) with a trace reflecting frequent transitions between individual light controls. The second rendering does not use all of the available space.

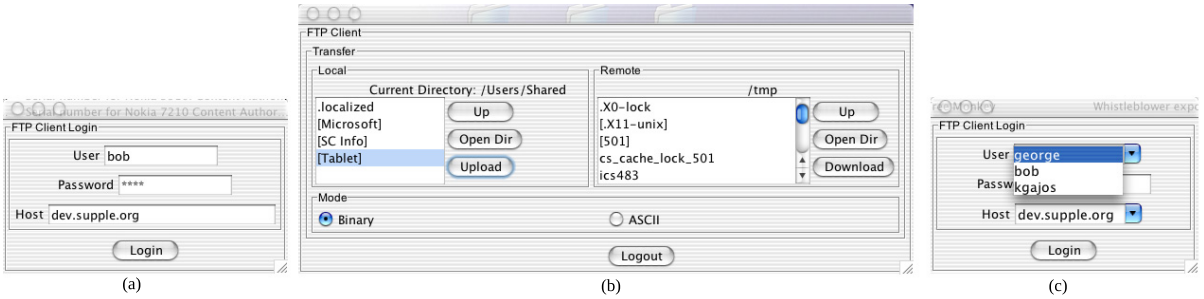

Figure 5: The interface for the FTP client application. (a) The initial login window. (b) The main window for transferring files between local and remote locations. (c) After logging out, the login window is rendered again – as the application adds user and host names to the respective lists of likely values, these changes are reflected in the new rendering of the login window.

5.4 An Interactive Interface

Figure 5 shows different windows rendered for the FTP client application. Figure 5(a) shows the initial login window. If the information entered is correct and the "Login" button is pressed, the login window is replaced by the main window (Figure 5(b)). Pressing the "Logout" button takes the user back to the login window. After each successful login, the application records the user names and host names entered by the user and adds them to the list of likely values for the corresponding state variables in the interface description. Thus, subsequent renditions of the login window reflect that change (Figure 5(c)). All of these windows are rendered and disposed of dynamically upon a request from the application, which supplies an abstract description to the renderer. In each case, the rendering takes less than a second.

5.5 Conceptual User Study

In order to compare SUPPLE's performance to that of human designers, we conducted an informal user study in which we asked human experts (graduate students in Computer Science who have taken at least one HCI course) to design user interfaces for the classroom under similar size and widget selection constraints as those imposed on SUPPLE. We gave our subjects a functional description of the appliances and a "library" of UI widgets. The widgets were printed on paper and cut out so the subjects' task was to arrange the widgets within a prescribed area. The tab pane widget was rendered as a booklet, thus allowing the subjects to fill the contents of each tab pane.

There were four subjects and two different size constraints. In the end we obtained three designs for each size constraint. Results for the size constraints depicted in Figure 4(a) varied the most and revealed interesting differences among our subjects as well as between the subjects and SUPPLE. Due to space constraints we present only a brief summary of our observations from this study.

One subject came up with a design which was almost identical to SUPPLE's, when its rendering was based on an empty user trace. Another subject felt that all light controls should be rendered together. Thus she rendered controls for all three lights in a single tab pane, using *spinner* widgets. She argued that users were likely to want to manipulate all lights at the same time and thus it was more important to render them next to each other than to use more convenient widgets such as sliders. We subsequently used the interface simulating the usage pattern hypothesized by that subject and recorded the trace. When we ran SUPPLE using the new user trace, we obtained a rendering equivalent to that produced by the subject (Figure 4(b)).

6. RELATED WORK

Our work follows two decades of research on model-based user interface design. In the terminology of Szekely's retrospective paper [18], SUPPLE falls into the category of *automated design tools*. Like other similar systems (*e.g.* [9, 11]), ours uses a *domain model* to describe the kinds and structure of the information to be exchanged between the user and the application. Unlike many of the other tools, SUPPLE does not have an explicit task or dialogue model, however the traces provide some information on the intended usage of the system.

Most importantly, however, the earlier work, including recent systems such as the Personal Universal Controller (PUC) [11], uses decision trees to render the interfaces. As noted by others [7], despite their advantages, decision trees have a number of problems. Most importantly, using this technique, interfaces are rendered in a single pass with all decisions taken at the level of individual widgets or groupings making effective tradeoffs between available screen size and widget quality nearly impossible. Another recent system, XWeb [12], is even further limited by the fact that leaf widgets are prespecified and only their layout is chosen dynamically.

Optimization-based, constraint techniques have a long history in user interface research, primarily for controling presentation and maintaining consistency (*e.g.* [4]). Recently, optimization has been used in new ways (*e.g.* LineDrive [1] and GADGET [8]). GADGET in particular has been used to lay out elements in a dialog box but it neither chose the widgets nor was capable of creating navigationally-complex interfaces (*i.e.*, ones that would not display all elements at once).

7. CONCLUSIONS AND FUTURE WORK

This paper makes four contributions: first, we formally define the interface rendering problem as a decision-theoretic optimization problem. Second, our framework includes as an input a trace of typical user behavior, allowing for user-specific renderings. Third, we present an efficient branch and bound rendering algorithm, implemented as the SUPPLE system. Fourth, we describe several experiments, showing the speed of SUPPLE, the quality of the resulting interfaces, and SUPPLE's ability to customize for individual users.

Despite our progress, much remains to be done. We need to explore the limits of our approach. We are extending SUPPLE to handle an even wider range of devices (e.g., wall-sized displays and J2ME phones). While we have demonstrated that SUPPLE is capable of generating interfaces for two different kinds of applications, they are both relatively simple. We aspire to complex applications such as Microsoft Word and Outlook, but this will require more sophisticated data models and attributes. We plan to remove Section 3's assumption that renditions are tree structured; this will allow SUPPLE to add shortcuts and duplicate commonly-used functionality. The approach will be to extend related work has been done for adaptive web sites [2, 3] and mobile portals [17]. The principle of *partitioned dynamicity* may alleviate user confusion [19]. We wish to make several extensions to the cost function: accounting for the actual old and new values in user traces and incorporating a number of established UI evaluation metrics. In particular, a metric like the *layout appropriateness* [16], which uses Fitts' Law and usage traces to evaluate the quality of precise widget layout within a pane, could help us to fine tune the rendering. Visual search models could also be used to evaluate the complexity of any given view of the interface; it has further been suggested that different visual search models could be used to reflect the behavior of users with different levels of expertise [5]. Finally, spatial information (e.g., the physical location of individual lights) could also be a useful addition to our interface specification, as shown in the iCrafter project [13].

Automatic interface adaptation complements explicit end-user customization, and we wish to investigate customiza-

tion languages and mechanisms. In particular, we would like to enable users to easily influence how SUPPLE interfaces are rendered. We are also exploring mechanisms for allowing users to customize the *functionality* of the interface rather than the presence and placement of pre-specified UI widgets.

Ultimately, the utility of our approach will be determined by adoption. We suspect that UI designers will be skeptical, uncomfortable with a functional specification, and preferring the fine control of layout provided by existing methods. A UI critiquing tool, based on SUPPLE, could still be useful to these designers, suggesting aspects of an interface that need review. And if end-users come to like automatic adaptation, because of trace-driven personalization or combination with explicit customization, then designers will be forced to take notice. The next step in this regard is a more detailed user study and evaluation.

8. ACKNOWLEDGEMENTS

The authors would like to thank the anonymous subjects who participated in our preliminary study. We would also like to acknowledge Mark Adler, Alan Borning, Gaetano Borriello, Tessa Lau, Jeffrey Nichols, Steven Wolfman and Alexander Yates for their thorough and insightfull comments on earlier drafts of this paper.

This research was funded in part by Office of Naval Research Grants N00014-98-1-0147 & N00014-02-1-0932, National Science Foundation Grants IRI-9303461, IIS-9872128, DL-9874759, and a CAREER Award.

9. REFERENCES

[1] M. Agrawala and C. Stolte. Rendering effective route maps: Improving usability through generalization. In E. Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 241–250. ACM Press / ACM SIGGRAPH, 2001.

[2] C. R. Anderson, P. Domingos, and D. S. Weld. Adaptive web navigation for wireless devices. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, 2001.

[3] C. R. Anderson and E. Horvitz. Web montage: a dynamic personalized start page. In *Proceedings of the eleventh international conference on World Wide Web*, pages 704–712. ACM Press, 2002.

[4] A. Borning and R. Duisberg. Constraint-based tools for building user interfaces. *ACM Transactions on Graphics (TOG)*, 5(4):345–374, 1986.

[5] A. Bunt, C. Conati, and J. McGrenere. What role can adaptive support play in an adaptive system. In *Proceedings of IUI'04*, Funchal, Portugal, 2004.

[6] L. Cardelli. Building user interfaces by direct manipulation. In *ACM Symposium on User Interface Software and Technology*, pages 152–166, 1988.

[7] J. Eisenstein, J. Vanderdonckt, and A. Puerta. Adapting to mobile contexts with user-interface modeling. In *Workshop on Mobile Computing Systems and Applications*, Monterey, CA, 2000.

[8] J. Fogarty and S. E. Hudson. GADGET: A toolkit for optimization-based approaches to interface and display generation. In *Proceedings of UIST'03*, Vancouver, Canada, 2003.

[9] W. C. Kim and J. D. Foley. Providing high-level control and expert assistance in the user interface presentation design. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 430–437, Amsterdam, The Netherlands, 1993. ACM Press.

[10] J. Lin and J. A. Landay. Damask: A tool for early-stage design and prototyping of multi-device user interfaces. In *In Proceedings of The 8th International Conference on Distributed Multimedia Systems (2002 International Workshop on Visual Computing)*, pages 573–580, 2002.

[11] J. Nichols, B. A. Myers, M. Higgins, J. Hughes, T. K. Harris, R. Rosenfeld, and M. Pignol. Generating remote control interfaces for complex appliances. In *CHI Letters: ACM Symposium on User Interface Software and Technology, UIST'02*, Paris, France, 2002.

[12] D. R. Olsen, S. Jefferies, T. Nielsen, W. Moyes, and P. Fredrickson. Cross-modal interaction using XWeb. In *Proceedings of the 13th annual ACM symposium on User interface software and technology*, pages 191–200, San Diego, California, United States, 2000.

[13] S. Ponnekanti, B. Lee, A. Fox, P. Hanrahan, and T. Winograd. ICrafter: A service framework for ubiquitous computing environments. In *Proceedings of Ubicomp 2001*, pages 56–75, 2001.

[14] A. Puerta and J. Eisenstein. XIML: A universal language for user interfaces, 2002. unpublished paper available at http://www.ximl.org/.

[15] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[16] A. Sears. Layout appropriateness: A metric for evaluating user interface widget layout. *Software Engineering*, 19(7):707–719, 1993.

[17] B. Smyth and P. Cotter. Personalized adaptive navigation for mobile portals. In *Proceedings of ECAI/PAIS'02*, Lyons, France, 2002.

[18] P. Szekely. Retrospective and challenges for model-based interface development. In F. Bodart and J. Vanderdonckt, editors, *Design, Specification and Verification of Interactive Systems '96*, pages 1–27, Wien, 1996. Springer-Verlag.

[19] D. S. Weld, C. Anderson, P. Domingos, O. Etzioni, K. Gajos, T. Lau, and S. Wolfman. Automatically personalizing user interfaces. In *IJCAI03*, Acapulco, Mexico, August 2003.

[20] A. Wexelblat and P. Maes. Footprints: History-rich tools for information foraging. In *CHI*, pages 270–277, 1999.

[21] C. Wiecha, W. Bennett, S. Boies, J. Gould, and S. Greene. ITS: a tool for rapidly developing interactive applications. *ACM Transactions on Information Systems (TOIS)*, 8(3):204–236, 1990.