Automatically Generating Personalized User Interfaces with Supple

Krzysztof Z. Gajos[a,b,*,1], Daniel S. Weld[a], Jacob O. Wobbrock[c]

[a]*Department of Computer Science and Engineering, Box 352350, University of Washington, Seattle, WA 98195, USA, tel. +1-206-543-9196*
[b]*Harvard School of Engineering and Applied Sciences, 33 Oxford St. Rm 251, Cambridge, MA 02138, USA, tel. +1-617-496-1876*
[c]*The Information School, Box 352840, University of Washington, Seattle, WA 98195, USA, tel. +1-206-616-2541*

Abstract

Today's computer-human interfaces are typically designed with the assumption that they are going to be used by an able-bodied person, who is using a typical set of input and output devices, who has typical perceptual and cognitive abilities, and who is sitting in a stable, warm environment. Any deviation from these assumptions may drastically hamper the person's effectiveness—not because of any *inherent* barrier to interaction, but because of a mismatch between the person's effective abilities and the assumptions underlying the interface design.

We argue that automatic personalized interface generation is a feasible and scalable solution to this challenge. We present our Supple system, which can automatically generate interfaces adapted to a person's devices, tasks, preferences, and abilities. In this paper we formally define interface generation as an optimization problem and demonstrate that, despite a large solution space (of up to 10^{17} possible interfaces), the problem is computationally feasible. In fact, for a particular class of cost functions, Supple produces exact solutions in under a second for most cases, and in a little over a minute in the worst case encountered, thus enabling run-time generation of user interfaces. We further show how several different design criteria can be expressed in the cost function, enabling different kinds of personalization. We also demonstrate how this approach enables extensive user- and system-initiated run-time adaptations to the interfaces after they have been generated.

Supple is not intended to replace human user interface designers—instead, it offers alternative user interfaces for those people whose devices, tasks, preferences, and abilities are not sufficiently addressed by the hand-crafted designs. Indeed, the results of our study show that, compared to manufacturers' defaults, interfaces automatically generated by Supple significantly improve speed, accuracy and satisfaction of people with motor impairments.

Key words: automatic user interface generation, optimization, adaptation, personalized user interfaces, ability-based user interfaces, Supple

1. Introduction

Today's computer-human interfaces are typically designed in the context of several assumptions: 1) that they are going to be used by an able-bodied individual, 2) who is using a typical set of input and output devices, 3) who has typical perceptual, cognitive, and motor abilities, and 4) who is sitting in a stable, warm environment. Any deviation from these assumptions (for example, hand tremor due to aging, using a mobile device with a multi-touch screen, low vision, or riding on a jostling bus) may drastically hamper the person's effectiveness—not because of any *inherent* barrier to interaction, but because of a mismatch between their effective abilities and the assumptions underlying the interface design.

This diversity of needs is generally ignored at the present time. Occasionally, it is addressed in one of several ways: manual redesign of the interface, limited customization support, or by supplying an external assistive technology. The first approach is clearly not scalable: new devices constantly enter the market, and people's abilities

[*]Corresponding author.

Email addresses: kgajos@eecs.harvard.edu (Krzysztof Z. Gajos), weld@cs.washington.edu (Daniel S. Weld), wobbrock@u.washington.edu (Jacob O. Wobbrock)

[1]Present address at Harvard University.

and preferences both differ greatly and often cannot be anticipated in advance [5]. Second, today's customization approaches typically only support changes to the organization of tool bars and menus, and cosmetic changes to other parts of the interface. Furthermore, even when given the opportunity, people do not customize [50, 63, 65], and even more rarely re-customize as their work habits change [52]. Finally, assistive technologies, while they often enable computer access for people who would otherwise not have it, also have limitations: assistive technologies can stigmatize their users; they are impractical for people with temporary impairments caused by injuries; they do not adapt to people whose abilities change over time; and finally, they are often abandoned, even by people who need them, because of factors like cost, complexity, configuration, and the need for ongoing maintenance [11, 13, 42, 68].

In contrast to these approaches, we argue that interfaces should instead be personalized to better suit the particular contexts of individual users. Many personalized interfaces are needed because of the myriad of distinct individuals, each with his or her own abilities, tasks, preferences, devices and needs. Therefore, traditional manual interface design and engineering will not scale to such a broad range of potential contexts and people. A different approach is needed. In this paper, we demonstrate that *automatic generation* of personalized user interfaces is a feasible and scalable solution to this challenge. We make the following specific contributions:

- We formally define interface generation as a discrete *constrained optimization* problem and solve it with a branch-and-bound algorithm using constraint propagation (Sections 3 and 4). This general approach allows our Supple system to automatically generate "optimal" user interfaces given a declarative description of an interface, device characteristics, available widgets, and a user- and device-specific cost function.

- We develop *two types of cost functions* for guiding the optimization process. The first is factored in a manner that enables preference-based personalization as well as fast computation, allowing Supple to generate user interfaces in under 1 second in most cases (Section 5.1). The second explicitly models a person's ability to control the pointer, allowing Supple to generate user interfaces adapted to unusual interaction techniques or abilities, such as an input jittery eye tracker or a user's limited range of motion due to a motor impairment (Section 5.2). Both types of cost functions incorporate usage traces, allowing Supple to generate interfaces that reflect a person's long-term usage patterns.

- We illustrate the extensibility of the approach by incorporating into the cost function a measure of *presentation consistency* among different variants of a user interface. This allows Supple to generate different user interfaces for an application such that these interfaces resemble one another, even if they are generated for different devices (Section 5.3).

- We demonstrate two approaches for *dynamic personalization* of Supple-generated user interfaces: an automatic system-driven adaptation to the current task, and a user-driven customization (Section 6).

- We systematically evaluate the systems issues in Supple and demonstrate that even for solution spaces on the order of 10^{17} possibilities, our algorithm can find the optimal rendering in less than a second in most cases. We also demonstrate that by exploring the solution space in parallel in two different orders, our algorithm's worst-case empirical performance can improve by up to two orders of magnitude (Section 7).

- We demonstrate a practical application of Supple: automatic generation of personalized user interfaces for people with motor impairments. The results of our user study show that user interfaces automatically generated by Supple can improve speed and accuracy for all users—and for people with motor impairments, they can also significantly improve satisfaction—compared to default user interfaces shipped with today's software (Section 8).

Automatic model-based user interface generation is frequently met with skepticism. Prior systems in this area—which, with few exceptions, attempted to incrementally improve existing interface design processes—were perceived to require higher up-front cost (learning a new language, manually building models) and to result in aesthetically less-pleasing artifacts than traditional, manual design approaches [56]. Instead, we believe that the real strength of automatic user interface generation lies not in incrementally improving existing design processes, but in enabling solutions to problems that cannot be adequately addressed by traditional methods. Our Supple system offers alternative user interfaces for those people whose individual devices, tasks, preferences, and abilities are not sufficiently addressed by the hand-crafted designs. In the case of people with motor impairments, the results of our study demonstrate that the benefits of personalized interfaces generated by Supple far outweigh the drawbacks of unfamiliar aesthetics. And, as Figure 33 illustrates, users with different sets of motor abilities benefit from very different user interface designs, suggesting that manual design methods would not scale.

We have previously presented fragments and refinements of this framework over the course of several years [21, 19, 22, 23, 27, 28]. This paper provides the first complete and consistent presentation of the technical underpinnings of the Supple system. The evaluation of the algorithm's performance and the parallel algorithm introduced in Section 7.4 have not been presented before.

2. Previous Research

Our Supple system automatically generates concrete user interfaces from declarative models that specify what types of information need to be exchanged between the application and the user. There have been a number of prior systems—such as COUSIN [34], Mickey [61], ITS [87], Jade [92], HUMANOID [80], UIDE [79], GENIUS [40], TRIDENT [83, 6], MASTERMIND [81], the "universal interaction" approach [36], XWeb [62], UIML [1], Personal Universal Controller [57] (and the related Huddle [59] and Uniform [58] projects), UI on the Fly [71], TERESA [67], Ubiquitous Interactor [60]—dating as far back as the 1980's, which used the model-based approach for user interface creation. The stated motivation for those prior efforts tended to address primarily two issues: simplification of the process of user interface creation and maintenance, and providing an infrastructure to allow applications to run on different platforms with different capabilities. In the case of earlier systems, the diversity of platforms was limited to different desktop systems, while more recent research (e.g., the "universal interaction" approach of [36], the Ubiquitous Interactor, TERESA) addressed the challenges of using dramatically different devices, such as phones, computers, touch screens, with very different sizes, input and output devices, and even modalities (such as graphical and voice). The authors of several of the earlier systems (for example, COUSIN, ITS, and GENIUS) also argued that their systems would help improve the consistency among different applications created for the same platform. A few (e.g., ITS and XWeb) also pointed out the potential of these systems for supporting different versions of the user interfaces adapted to the special needs of people with impairments, but none of these projects resulted in any concrete solutions for such users. In summary, prior research was primarily motivated by the desire to improve the existing user interface-development practice. The Huddle system was a notable exception, in that it provided automatically generated user interfaces for dynamically assembled *collections* of connected audio-visual appliances, such as personal home theater setups. In those systems, the available functionality depends on the selection of appliances and the connections among them, and can change over time as the components are replaced. Thus, by automatically generating interfaces for these often unique and evolving systems, Huddle provided novel capability that would not have been available using existing interface-design methods. Although a similar approach was earlier proposed by the iCrafter project [69], Huddle was the first to provide a complete implementation that included an interface-generation capability.

The level of automation provided by the previous systems varied from providing just the appropriate programmatic abstractions (e.g., UIML), to design tools (e.g., COUSIN), to mixed-initiative systems providing partially automated assistance to the programmer or the designer (e.g., TRIDENT, TERESA). Very few systems considered fully-autonomous, run-time generation of user interfaces, and of those only the Personal Universal Controller [57] (and the related Huddle and Uniform projects) resulted in a complete system while others (e.g., the "universal interaction" approach [36] or XWeb) assumed the existence of an external interface generator.

Of those systems which provided some mechanism to automatically generate user interfaces, the majority used a simple rule-based approach, where each type of data was matched with precisely one type of interactor that would be used to represent it in the user interface (e.g., Mickey, ITS, GENIUS, the Ubiquitous Interactor). TRIDENT was probably the first system to take more complex context information into account when generating user interfaces. For example, it explicitly considered whether the range of possible values represented by a selector would be allowed to change at run time, whether a particular number selection would be done over a continuous or discrete range, the interaction between interface complexity and the available screen space, as well as the expected user expertise. As a result, TRIDENT required a much more complex rule base than its predecessors—eventually the authors collected a set of 3700 rules [82] represented as a decision tree. The Personal Universal Controller system also takes into account rich context but by limiting the domain of interfaces to appliance controllers it did not require as large a knowledge base as TRIDENT.

In terms of their approach to abstractly representing user interfaces, most systems relied on a type-based declarative model of the information to be exchanged through the interface, as well as on some information about how different elements were grouped together. Often these two kinds of information were combined together into a single hierarchical model, which in recent systems is often referred to as the Abstract User Interface (AUI) [67]. In many cases, the interface model was specified explicitly (e.g, Personal Universal Controller, TERESA, UIML), while in some systems it was inferred from the application code (e.g., in Mickey, HUMANOID) or from a database schema (GENIUS). A number of the systems also included a higher-level task or dialogue model. For example, GENIUS represented interaction dynamics through the Dialogue Nets, TRIDENT relied on Activity Chaining Graphs, MASTERMIND modeled tasks in terms of goals and pre-conditions, while TERESA used hierarchical ConcurTaskTrees [66].

Constraints have been used as a way to define flexible layouts which provided some level of device independence [7, 8]. In those systems, semantically meaningful spatial relationships among user interface elements could be encoded as constraints, and—if a feasible solution existed—the constraint solver would generate an arrangement that satisfied all the constraints.

Constrained optimization subsumes the constraint satisfaction approaches in that it produces the *best* result that satisfies the constraints. Optimization-based techniques are being increasingly used for dynamically creating aspects of information presentation and interactive systems. For example, LineDrive system [3] uses optimization to generate driving maps that emphasize the most relevant information for any particular route. The Kandinsky system [17] creates information visualizations that mimic the styles of several visual artists. The RIA project uses an optimization-based approach to select what information to present to the user [93], and how to best match different pieces of information to different modalities [94]. Optimization is also a natural technique for automatically positioning labels in complex diagrams and visualizations [85]. Motivated by the growing use of optimization in automating parts of the interactive systems, the GADGET toolkit [18] provides a general framework for incorporating optimization into interactive systems, and it has been used to reproduce the LineDrive functionality and to automatically generate user interface layouts.

Before SUPPLE, optimization was used for graphical user interface generation by the GADGET toolkit and with the Layout Appropriateness user interface quality metric [76]. In both cases, optimization was used to automatically generate the user interface layout. In contrast, SUPPLE uses a single constrained optimization procedure to generate the layout but also to select the appropriate interactors for different user interface elements, and to divide the interface into navigational components, such as windows, tab panes, popup windows, etc. When generating user interfaces adapted to a person's motor abilities, SUPPLE also uses the same optimization procedure to find the optimal size for all the clickable elements in the interface, thus solving a much harder problem than those attempted in prior work.

3. Representing Interfaces, Devices and Users

Like other automatic user interface generation systems, SUPPLE relies on an *interface specification* ($\mathcal{I}$). Additionally, SUPPLE also uses an explicit *device model* ($\mathcal{D}$) to describe the capabilities and limitations of the platform for which the interface is to be generated. Finally, in order to reflect individual differences among usage patterns, SUPPLE additionally includes a *usage model*, represented in terms of *user traces* ($\mathcal{T}$). We describe each of these components below.

3.1. Functional Interface Specification ($\mathcal{I}$)

SUPPLE adopts a functional representation of user interfaces—that is, one that says *what* functionality the interface should expose to the user instead of *how* to present those features. Like a number of previous systems (e.g., [1, 57, 62]), SUPPLE represents basic functionality in terms of types of data that need to be exchanged between the application and the user. Semantic groupings of basic elements are expressed through *container types*, which also serve as reusable abstractions. This is in contrast to several other systems that use task-oriented specification languages (e.g., [67, 81]), which try to capture the logical activities performed with the user interface by representing not only user interface objects, but also the dependencies among them. By specifying user interfaces at a higher level of abstraction, task-oriented languages allow for greater flexibility in generating concrete user interfaces from any abstract specification.

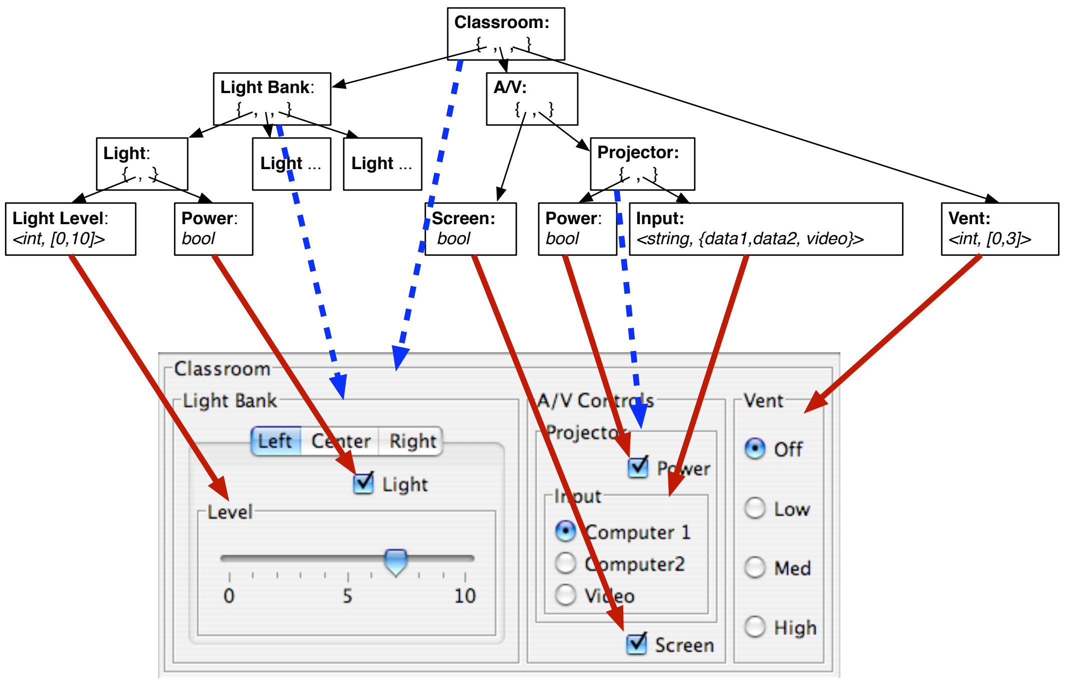


Figure 1: Supple uses an algorithm that makes discrete assignments of widgets to the elements of the functional interface specification. This figure illustrates a functional specification and a sample concrete user interface for an application controlling a small set of devices in a classroom. The solid arrows show the assignments of primitive widgets to the elements of the interface specification corresponding to the actual functionality in the underlying application. The dashed arrows show the assignments of container widgets to the intermediate nodes in the specification; for example the Light Bank is rendered as a tab pane while the projector was assigned a vertical layout.

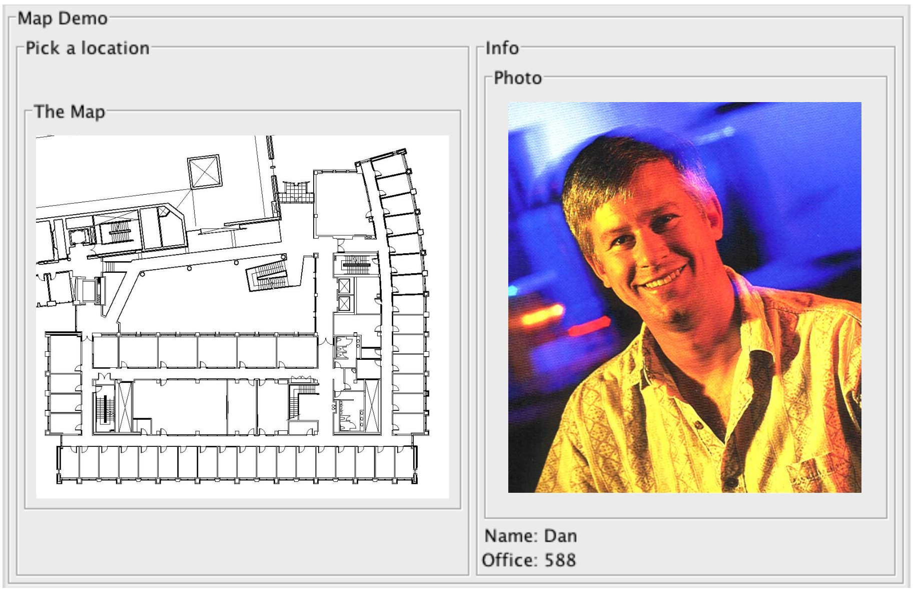
For example, a hotel reservation interface can be instantiated as a step-by-step wizard for novice users or as a single view for hotel registration staff and travel agents. We chose not to adopt this task-oriented approach for two reasons. First, because task-oriented descriptions are typically first compiled into a data-oriented functional description [67], our use of a functional specification does not preclude a future use with a task-oriented system. Second, task-oriented languages are particularly useful for capturing task-oriented processes such as store checkout or making a hotel reservation. Most direct manipulation systems, however, support a broad range of possible tasks and make simultaneously available numerous reversible actions. Such interfaces would not benefit significantly from a task-oriented representation.

To illustrate our approach, the upper part of Figure 1 shows the formal specification of the interface for a simple application for controlling lighting, ventilation, and audiovisual equipment in a classroom. Formally, an interface is defined to be $\mathcal{I} \equiv \langle \mathcal{S}_f, \mathcal{C}_\mathcal{I} \rangle$, where $\mathcal{S}_f$ is a tree of *interface elements*, and $\mathcal{C}_\mathcal{I}$ is a set of *interface constraints* specified either by the designer at design time, or by the user at run time through Supple's customization mechanism (Section 6.2).

The interface elements included in the functional specification correspond to units of information that need to be conveyed via the interface between the user and the controlled appliance or application. The interface constraints can, in principle, constrain any aspect of interface presentation. In practice, we rely on the following three classes of constraints:

- equality constraints, which allow multiple instances of the same type (for example, all three lights in the Classroom interface in Figure 1) to be rendered identically;

- constraints limiting the set of presentation options for an element, which allow the user, for example, to use the customization mechanism to constrain light intensity to be rendered with a slider or to forbid the use of tab panes at the top level of the Classroom interface;



Name: Dan
Office: 588

Figure 2: An interface utilizing images and clickable maps.

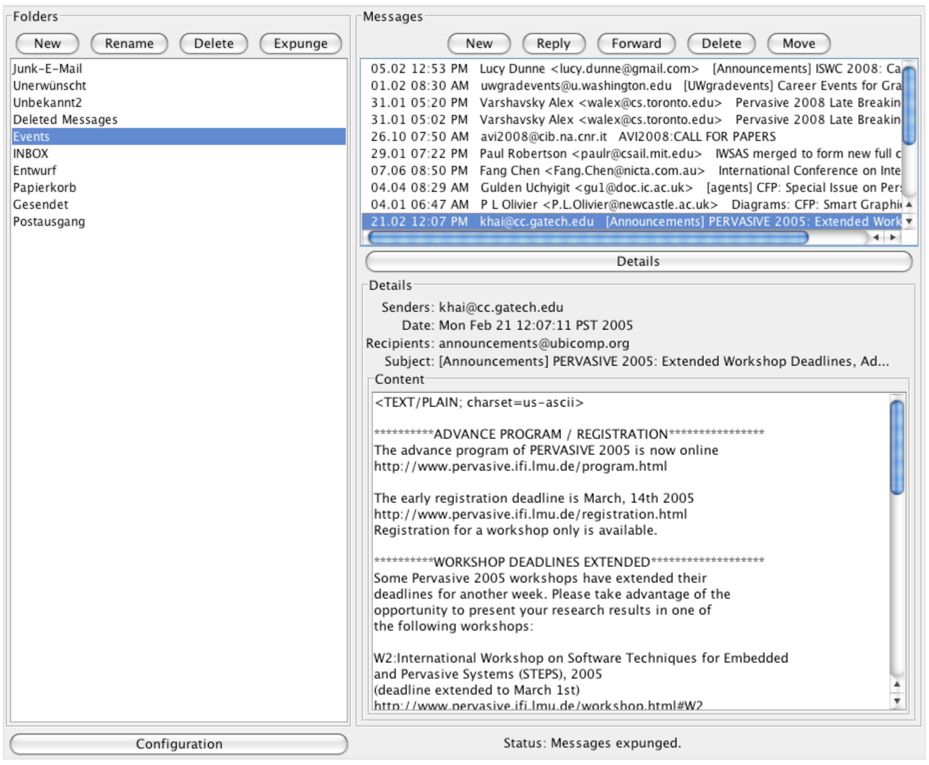
- interdependence constraints (for example, a stylistic requirement that a checkbox cannot be rendered as the sole element inside a tab pane).

The elements in the functional specification are defined in terms of their type. There are several classes of types:

Primitive types include the common basic data types such as integers, floats, strings and booleans. As an example, the power switches for the lights are represented as booleans in the specification of Figure 1. Primitive types also include several more specialized constructs that often benefit from special handling by user interfaces, such as dates, times, images and clickable maps. These last two types are illustrated in a concrete interface for an interactive map application shown in Figure 2, where a person can point at different offices on a building map, causing the occupant's image to be displayed in the panel on the right-hand side. Some primitive types can be further described with a small number of attributes. For example, information about the expected length of input can be added to instances of string type.

Container types, formally represented as $\{\tau_1, \tau_2, \ldots, \tau_n\}$, are used to create groups (or records) of simpler elements, τ_i. For example, all of the interior nodes (e.g., Classroom, Light Bank, Light, etc.) in the specification tree in Figure 1 are instances of the container type. The container types serve two functions. First, they provide SUPPLE with information as to what pieces of functionality belong together semantically. Secondly, they provide reusable abstractions: as with all SUPPLE types, a container type can be specified once and later instantiated in multiple parts of the interface.

Constrained types: $\langle \tau, C_\tau \rangle$ denotes a constrained type, where τ is any primitive or container type and C_τ is a set of constraints over the values of this type. In the classroom example, the light level is defined as an integer type whose values are constrained to lie between 0 and 10. In the email client shown in Figure 3a, the list of email folders shown on the left is represented as a string whose values are constrained to be the names of the folders in the currently selected email account. Constraints can also be specified for container types. For example, consider the list of available email accounts in the email example of Figure 3b. Each account is modeled as an instance of the container type. Yet the user wants not only to see the settings of a single account, but also wants to select different accounts to view. Thus, the interface element representing the current account is modeled as a container object whose domain of values is restricted to registered email accounts for that user. When SUPPLE renders this container, it allows the user to select which account to view, and also displays that account's settings. When enough screen space is available, SUPPLE will render both the selection mechanism and the details of the content side-by-side, as in Figure 3b. When space is



Junk-E-Mail
Unerwünscht
Unbekannt2
Deleted Messages
Events
INBOX
Entwurf
Papierkorb
Gesendet
Postausgang

```
05.02 12:53 PM   Lucy Dunne <lucy.dunne@gmail.com>   [Announcements] ISWC 2008: Ca
01.02 08:30 AM   uwgradevents@u.washington.edu   [UWgradevents] Career Events for Gra
31.01 05:20 PM   Varshavsky Alex <walex@cs.toronto.edu>   Pervasive 2008 Late Breakin
31.01 05:02 PM   Varshavsky Alex <walex@cs.toronto.edu>   Pervasive 2008 Late Breakin
26.10 07:50 AM   avi2008@cib.na.cnr.it   AVI2008:CALL FOR PAPERS
29.01 07:22 PM   Paul Robertson <paulr@csail.mit.edu>   IWSAS merged to form new full c
07.06 08:50 PM   Fang Chen <Fang.Chen@nicta.com.au>   International Conference on Inte
04.04 08:29 AM   Gulden Uchyigit <gu1@doc.ic.ac.uk>   [agents] CFP: Special Issue on Per
04.01 06:47 AM   P L Olivier <P.L.Olivier@newcastle.ac.uk>   Diagrams: CFP: Smart Graphi
21.02 12:07 PM   khai@cc.gatech.edu   [Announcements] PERVASIVE 2005: Extended Work
```

Details

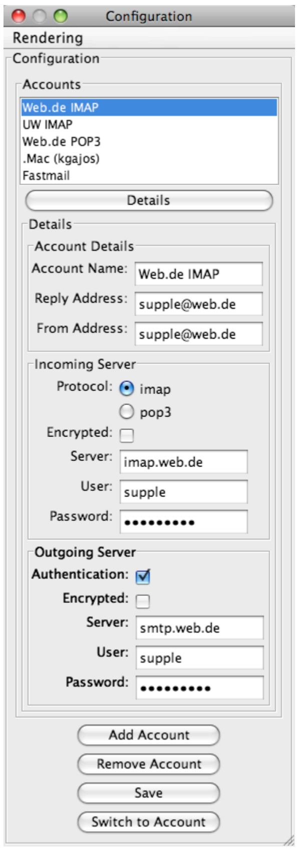
Senders: khai@cc.gatech.edu
Date: Mon Feb 21 12:07:11 PST 2005
Recipients: announcements@ubicomp.org
Subject: [Announcements] PERVASIVE 2005: Extended Workshop Deadlines, Ad...

Content

```
<TEXT/PLAIN; charset=us-ascii>

**********ADVANCE PROGRAM / REGISTRATION****************
The advance program of PERVASIVE 2005 is now online
http://www.pervasive.ifi.lmu.de/program.html

The early registration deadline is March, 14th 2005
http://www.pervasive.ifi.lmu.de/registration.html
Registration for a workshop only is available.

**********WORKSHOP DEADLINES EXTENDED******************
Some Pervasive 2005 workshops have extended their
deadlines for another week. Please take advantage of the
opportunity to present your research results in one of
the following workshops:

W2:International Workshop on Software Techniques for Embedded
and Pervasive Systems (STEPS), 2005
(deadline extended to March 1st)
http://www.pervasive.ifi.lmu.de/workshop.html#W2
```

Status: Messages expunged.

(a)



Web.de IMAP
UW IMAP
Web.de POP3
.Mac (kgajos)
Fastmail

Account Name: Web.de IMAP
Reply Address: supple@web.de
From Address: supple@web.de

Protocol: [✓ imap] pop3
Encrypted: ☐
Server: imap.web.de
User: supple
Password: •••••••••

Authentication: ☑
Encrypted: ☐
Server: smtp.web.de
User: supple
Password: •••••••••

(b)

Figure 3: An email client that uses Supple to render its user interface. (a) The main view. (b) The configuration pane.

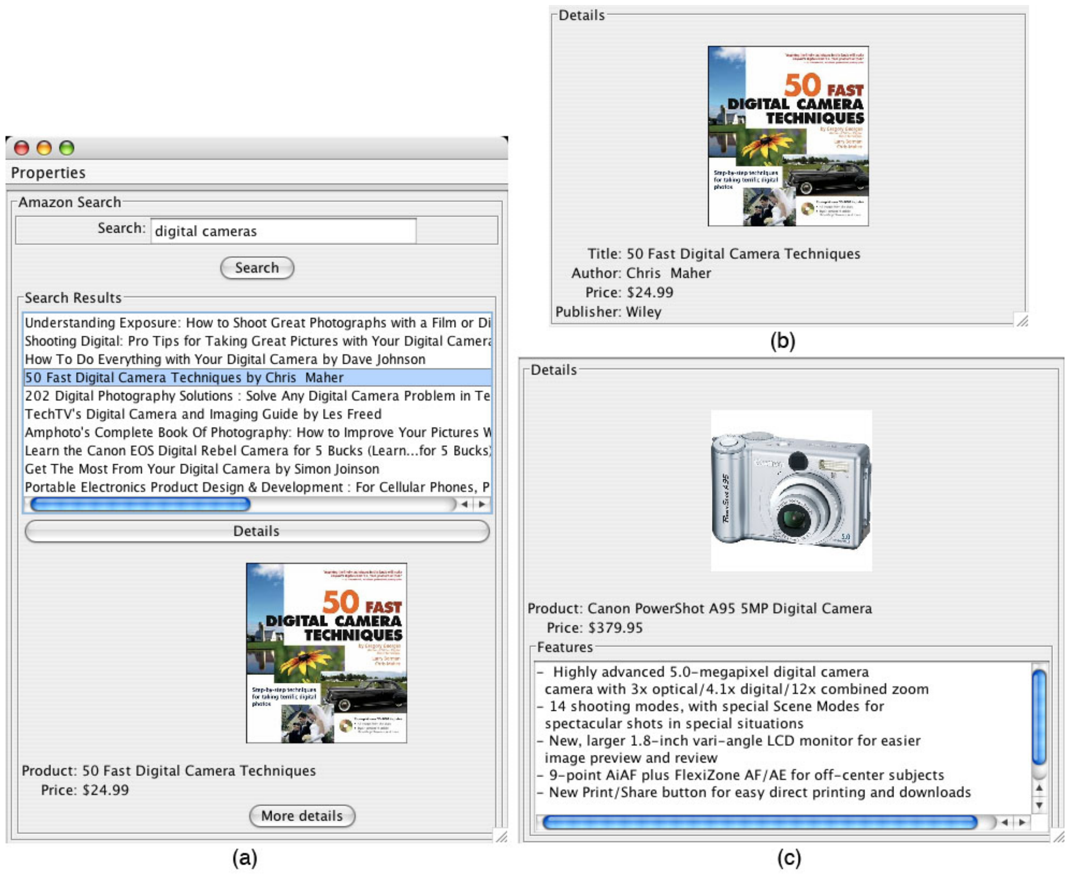
scarce, Supple will show just the list of available accounts; in order to view their contents, the user must double-click on an element in the list, or click the explicit "Details" button.

The constraints can be of any type, but typically they are expressed as an enumeration of allowed values or as a range. Further, the constraints on the legal values of an element are allowed to change dynamically at run time—for example, the list of folders from which to select will change when folders are created or deleted. Additional relevant attributes can be specified in a definition of a constrained type, such as whether the constraint can change at run time or not, or what the expected size of the domain of possible choices is.

The elements of the constrained type are often rendered with discrete selection widgets such as lists, radio buttons, or combo boxes. But they can also be rendered as sliders for continuous number ranges where precise selection is not required, or even as validated edit boxes.

A number of previous interface description languages, such as those used in the Personal Universal Controller [57] or TERESA [67] projects, explicitly distinguish between types that can be manipulated with selection versus text editing operations. However, in some situations, both interactions may be appropriate. For example, selecting a number from a small range can be represented as a list (selection) or as a validated input (edit). With the constrained types, Supple avoids making this commitment.

Subtyping. While the above approach makes modeling easy, it assumes that for constrained container types, all the possible values allowed by the constraint are of the same type. In practice, this is not always the case. For example, consider the interface to Amazon Web Services in Figure 4. Items returned by search may come from any of several categories, each of which can have different attributes. Books, for example, have titles and authors, while many other



(a)

Properties

Amazon Search

Search: digital cameras

Search

Search Results

Understanding Exposure: How to Shoot Great Photographs with a Film or Di
Shooting Digital: Pro Tips for Taking Great Pictures with Your Digital Camera
How To Do Everything with Your Digital Camera by Dave Johnson
50 Fast Digital Camera Techniques by Chris Maher
202 Digital Photography Solutions : Solve Any Digital Camera Problem in Te
TechTV's Digital Camera and Imaging Guide by Les Freed
Amphoto's Complete Book Of Photography: How to Improve Your Pictures W
Learn the Canon EOS Digital Rebel Camera for 5 Bucks (Learn...for 5 Bucks)
Get The Most From Your Digital Camera by Simon Joinson
Portable Electronics Product Design & Development : For Cellular Phones, P

Details

Product: 50 Fast Digital Camera Techniques
Price: $24.99

More details

(b)

Details

Title: 50 Fast Digital Camera Techniques
Author: Chris Maher
Price: $24.99
Publisher: Wiley

(c)

Details

Product: Canon PowerShot A95 5MP Digital Camera
Price: $379.95

Features

- Highly advanced 5.0-megapixel digital camera
camera with 3x optical/4.1x digital/12x combined zoom
- 14 shooting modes, with special Scene Modes for
spectacular shots in special situations
- New, larger 1.8-inch vari-angle LCD monitor for easier
image preview and review
- 9-point AiAF plus FlexiZone AF/AE for off-center subjects
- New Print/Share button for easy direct printing and downloads

Figure 4: A simple client for Amazon Web Services. (a) Search results with a pane showing properties of a selected object. Only those properties which are common to all items are shown there, but the "More Details" button brings up a specialized view for each item. (b) Detailed view for a book. (c) Detailed view for a digital camera.

items do not. To alleviate this problem, Supple allows the elements of a container of type τ to be a subtype[2] τ' of τ. The Amazon Web Services example in Figure 4 illustrates one way subtypes can be rendered in a concrete graphical user interface: if space permits, Supple renders all the attributes of the common ancestor type τ statically, next to the choice element (Figure 4a). Any time a specialized object is selected by the user, another button is highlighted, alerting the user that more detailed information is available, which can be displayed in a separate window as shown in Figures 4b and 4c.

Vectors: elements of type $vector(\langle\tau, C_\tau\rangle)$ denote an ordered sequence of zero or more values of type τ and are used to support multiple selection. Like in the constrained types, the constraints C_τ define the set of values of type τ that can be selected. For example, the list of emails in the email client (Figure 3a) is represented as a vector of Message elements, whose values are constrained to the messages in the currently selected folder; this allows the user to select and move or delete several messages at once.

Actions are denoted with a functional type signature, $\tau_1 \mapsto \tau_2$, where τ_1 stands for the type of the object containing parameters of the action and τ_2 describes the return type, that is, the interface component that is to be displayed after the typical execution of the action. Unlike the other types which are used to represent an application's *state*, the action type is used to invoke an application's *methods*. For example, the Login button in the FTP Client Login interface (Figure 5a) is represented as an action. Its parameter is a container holding the User, Password, and Host elements,

[2]A subtype of a container type is created by adding zero or more new elements; the subtype cannot rename, remove, or change the type of elements defined in its parent type.

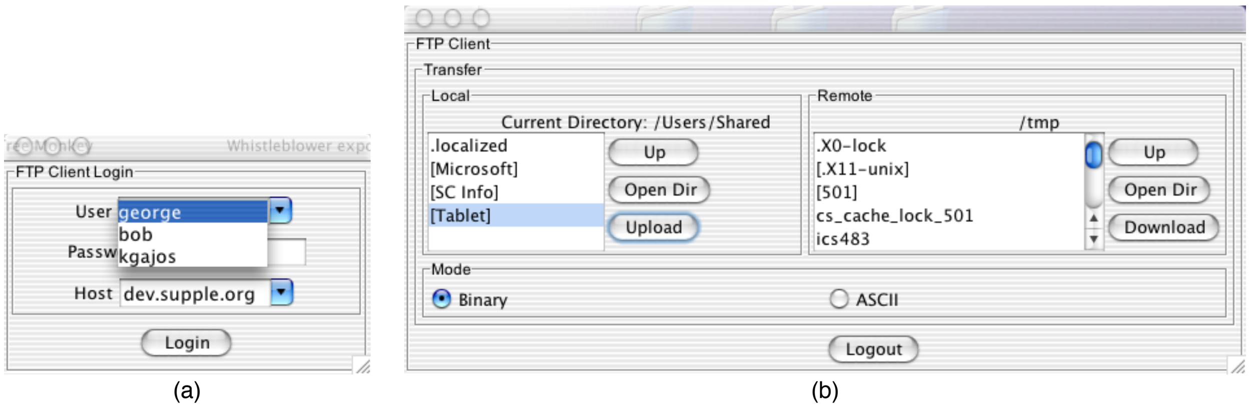


Figure 5: This simple FTP client UI illustrates the Action type in SUPPLE's functional specification.

while its output is the container type describing the FTP Client interface (Figure 5b), which appears after the successful execution of a login action.

The parameters and the return type of an action type can be *null* if the action has no parameters or causes no new interface elements to be created. For example, the New action in the Email client (Figure 3a) has *null* parameter type, and the Search action in the Amazon Web Services client (Figure 4a) has *null* return type because it only alters the contents of the existing Search Results part of the existing interface.

3.2. Device Capabilities and Constraints ($\mathcal{D}$)

We model a display-based device as a tuple:

$$\mathcal{D} \equiv \langle \mathcal{W}, C_{\mathcal{D}} \rangle$$

where $\mathcal{W}$ is the set of available user interface widgets on that device and $C_{\mathcal{D}}$ denotes a set of device-specific constraints.

Widgets are objects that can turn elements from the functional specification into components of a rendered interface. 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 panels, tab panes, etc.).

Like the interface constraints, the device-specific constraints in $C_{\mathcal{D}}$ 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 check whether the interface exceeds the available screen size.

Common widget toolkits are often a poor fit for unusual interactions (e.g., trying to control a mouse cursor with a laser pointer) or abilities (e.g., for people with impaired dexterity or low vision). To accommodate such unusual interactions and abilities, we extended one standard widget toolkit in two ways: by adding new widgets and by parametrizing each widget with two continuous parameters, the *minimum target size*, s_t, and the *minimum visual cue size*, s_c. The minimum target size parameter—used only on devices that support 2D pointer control—constrains the minimum size of any widget component that can be controlled with a pointer. Examples include a button, a list element, or a slider, as illustrated in the left pane of Figure 6. The minimum visual cue size constrains the size of important visual cues, such as fonts and icons (the right pane of Figure 6).

The new widgets (see Figure 7), which provide alternatives to a checkbox, a set of radio buttons, and a spinner, expand SUPPLE's options when generating user interfaces for touch-based interactions and for situations where users' dexterity is impaired due to context of use or due to a health condition.

3.3. Modeling Users with Traces ($\mathcal{T}$)

Most people use only a small subset of functions available in any application, and different users use different subsets [29, 44]. To adapt to a person's tasks and long-term usage patterns, the user interface should be rendered such that important functionality is easy to manipulate and to navigate to. Instead of relying on explicit annotations by the

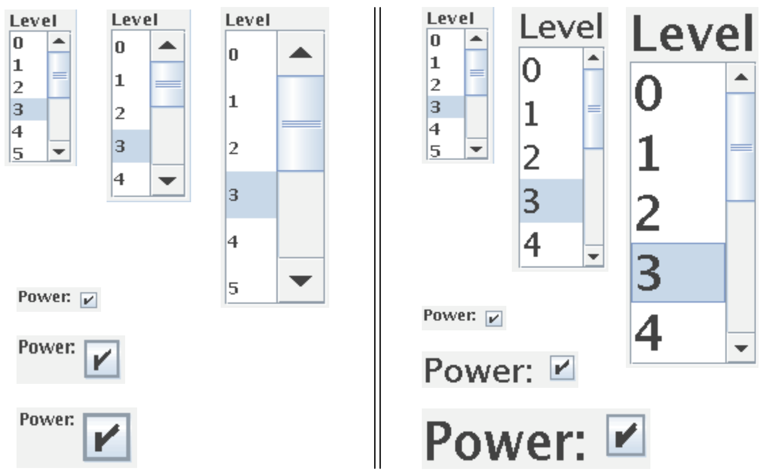


Figure 6: Presentation of a list widget and a checkbox widget for different values of (left) the minimum target size s_t, and (right) the minimum visual cue size s_c parameters.

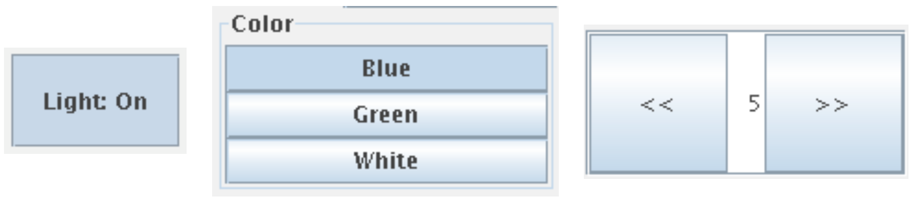


Figure 7: We have extended a standard widget toolkit with three additional widgets, to use as alternatives to (left) a checkbox, (center) a set of radio buttons, and (right) a spinner.

designer or the user, Supple relies on usage traces, which can correspond either to actual or anticipated usage. Usage traces provide not just interaction frequency for primitive widgets, but also frequencies of transitions among different interface elements. In the context of the optimization framework, traces offer the possibility of computing *expected* cost with respect to anticipated use.

A usage trace, $\mathcal{T}$, is a set of *trails* where, following [86], the term trail refers to "coherent" sequences of elements manipulated by the user (i.e., the abstract elements from the interface description and not the widgets used for rendering). 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 values this element assumed (if appropriate). It is further assumed that $u_0 = \langle root, -, - \rangle$, where *root* stands for the root element in the functional specification tree.

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. Note that in some cases, use of different devices may be correlated with different contexts of use (for example, a person may mix and organize music on a desktop computer, but primarily use the playback functionality while traveling with a mobile device), which is why the sharing of usage traces across platforms is optional.

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 will enable the system to work correctly with empty or sparse user traces. Also, 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.

Finally, while it may be conceptually helpful to think of modeling users in terms of actual traces, those traces can grow arbitrarily large. Therefore, in Section 5 we will show that Supple only needs to maintain concise summary statistics to adapt to a particular pattern of usage.

4. Interface Generation as Optimization

The goal is to render each interface element with a concrete widget, as illustrated earlier in Figure 1. Thus a *legal rendering* of a functional specification $\mathcal{S}_f$ is defined to be a mapping $R : \mathcal{S}_f \mapsto \mathcal{W}$ which satisfies the interface and device constraints in $C_{\mathcal{I}}$ and $C_{\mathcal{D}}$. Of course, there may be many legal renderings. Therefore, in order to find the best one, Supple relies on a *cost function* $\$: R \mapsto \mathbb{R}_{\geq 0}$, which provides a quantitative metric of the user interface quality. The cost function can correspond to any measure of quality of a user interface, such as consistency with the user's stated preferences (Section 5.1) or expected speed of use (Section 5.2). It can also incorporate additional concerns, such as similarity to previously seen renderings of a user interface, even if those renderings were generated for other devices (Section 5.3).

We thus formally define the *interface rendering problem* as a tuple $\langle \mathcal{I}, \mathcal{D}, \mathcal{T}, \$ \rangle$, where $\mathcal{I} \equiv \langle \mathcal{S}_f, C_{\mathcal{I}} \rangle$ abstractly describes the interface in terms of the functional specification and the interface constraints, $\mathcal{D} \equiv \langle \mathcal{W}, C_{\mathcal{D}} \rangle$ is a device model specifying available widgets and device constraints, $\mathcal{T}$ is the usage trace, and $\$$ is the cost function. R is a *solution* to a rendering problem if R is a legal rendering with minimum cost—we thus cast interface generation as constrained optimization, where the goal is to find a concrete user interface that minimizes the expected value of the cost function with respect to the usage trace, subject to the interface and device constraints. As stated, this is a hard discrete/continuous hybrid problem because $\mathcal{W}$ contains different classes of widgets, each of which is parametrized with several real parameters, such as minimum target size s_t, minimum visual cue size s_c, and additional widget-specific parameters, for example, the length of a list widget for showing search results in the Amazon search interface (Figure 4), can vary reasonably from a handful up to 40 entries.

Regardless of the particular cost function used, the cost of the best user interface is not likely to be a monotonic or even a continuous function of the minimum target size s_t or the minimum visual cue size s_c. This is because of the screen size constraint: as larger and larger widget sizes are used in response to changes to s_t or s_c, the available amount of screen space will eventually be exceeded, making it necessary to use more compact widgets (e.g., a combo box instead of a list) or different navigation strategies (e.g., tab panes instead of a side-by-side layout). For this reason, one cannot apply any of the efficient convex optimization techniques. Instead, it is necessary to search the space exhaustively. Fortunately, the specter of continuous optimization is only an illusion because in practice only integer sizes are used. Furthermore, one may approximate matters by discretizing the space even more coarsely—for example, at 5 pixel intervals—yielding 21 discrete settings (in the range between 0 and 100) for the size parameter. This allows us to cast the problem as *constrained discrete optimization*.

Conceptually, Supple enumerates all possible interfaces for a particular application and chooses the one which minimizes the user's expected cost of interaction. To find a globally optimal solution, we use an algorithm that combines branch-and-bound search [47, 55] with constraint satisfaction techniques. This algorithm is illustrated at a high level in Table 1, where the *variables* correspond to the elements in the functional specification $\mathcal{S}_f$, their possible *values* are drawn from the set of available widgets $\mathcal{W}$, and the *constraints* include both interface and device constraints (i.e., $C_{\mathcal{I}}$ and $C_{\mathcal{D}}$). Efficiency of this algorithm is affected by several design choices:

- the admissible heuristic (the *estimatedSolutionCost* function on line 2), which helps prune provably suboptimal solutions,
- the constraint propagation strategy (the *propagateConstraints* function on line 1), which helps eliminate provably illegal solutions,
- the variable and value ordering heuristics (the *selectUnassignedVariable* function on line 7, and the *orderValues* function on line 8).

We discuss these choices in turn.

4.1. The Admissible Heuristic

At each step of the search process, an admissible heuristic provides a lower bound on the cost of the best solution given the partial choices made so far. The closer the heuristic approximates the cost of the actual best solution reachable from a given point in the search process, the more effective it is at pruning sub-optimal solutions and, hence, the faster the algorithm. The form of the admissible heuristic depends on the form of the cost function. In the next section, we derive two cost functions and corresponding admissible heuristics.

```
bestCost ← ∞
bestRendition ← null

function optimumSearch(variables, constraints)
1.  if propagateConstraints(variables, constraints) = fail
        return
2.  if estimatedSolutionCost(variables) ≥ bestCost
        return
3.  if completeAssignment(variables) do
4.      bestCost ← cost
5.      bestRendition ← variables
6.      return
7.  var ← selectUnassignedVariable(variables)
8.  for each value in orderValues(getValues(var))
9.      setValue(var, value)
10.     optimumSearch(variables, constraints)
11. restoreDomain(var)
12. undoPropagateConstraints(variables)
13. return
```

Table 1: An algorithm combining branch-and-bound discrete optimization and constraint satisfaction mechanisms. The *variables* correspond to the elements in the functional specification $\mathcal{S}_f$, their possible *values* are drawn from the set of available widgets $\mathcal{W}$, and the *constraints* include both interface and device constraints (i.e., $C_{\mathcal{I}}$ and $C_{\mathcal{D}}$). The solution is stored in *bestRendition*.

4.2. *Constraint Propagation*

We have further optimized the algorithm by implementing full constraint propagation for size constraints 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 size constraints. This allows the algorithm to more quickly detect paths that will not yield a legal solution. Furthermore, it allows the admissible heuristics to make more accurate estimates of the final cost of the complete interface allowing for more efficient branch-and-bound pruning.

In general, full constraint propagation requires time that is quadratic in the number of variables [73]. Note, however, that widget size constraints form a tree structure that mirrors the hierarchy of the functional specification. Exploiting this, Supple performs full propagation of size constraints in linear time. The other types of constraints can form a potentially arbitrary network and Supple uses a one-step forward checking procedure (i.e., propagation of constraints only to the immediate neighbors) for those constraints. The evaluation of the system's performance (Section 7.4) shows that these optimizations are indeed very effective.

4.3. *Variable Ordering*

The search is directed by the variable ordering scheme encoded in the *selectUnassignedVariable* subroutine. Because all variables are eventually considered, 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* first chooses the leaf variables 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. The final heuristic, *minimum remaining values* (MRV), has proven highly effective in many constraint satisfaction problems [73]; the idea is always to focus on the most constrained variable, that is, the one with the fewest possible values remaining.

4.4. *Value Ordering*

Ordering of values for each variable is done in a greedy manner, with those with minimum marginal cost being tried first. While other approaches are common in solving constraint satisfaction problems, we are concerned with finding the best possible interface. In practice, when the problem is under-constrained, this leads to efficient selection of the best solution while in over-constrained cases, the constraint propagation procedure efficiently eliminates low-cost but large widgets from among the possible values.

5. Formulating the Cost Function

The style and quality of user interfaces generated by Supple is determined by the cost function, which provides a quantitative metric of user interface quality. In this section, we develop two different cost functions. The first is factored in a manner that enables fast computation of an admissible heuristic. This cost function is also parametrized in such a way that different choices of parameters can result in different styles of user interfaces generated. Subsequent work [28] demonstrates a preference elicitation approach that allows this cost function formulation to capture a user's *subjective* preferences regarding how his or her user interfaces should be generated. The second cost function (Section 5.2) reflects the expected time a person would need to perform a typical set of tasks with a particular user interface. This cost function can capture a person's *objective* motor abilities [27] and allows user interfaces to be directly optimized for speed of use. The last part of this section describes an extension that enables a notion of presentation consistency to be included as one of the terms in the cost function.

5.1. *Factorization for Efficient Computation and Personalization*

To develop a cost function that supports fast performance of the optimization algorithm as well as personalization, we start with three design requirements:

1. As discussed in Section 3.3, to enable generating user interfaces adapted to a person's usage patterns, the cost function should take into account information from usage traces, so as to provide an estimate of the *expected* cost with respect to the actual or anticipated usage. This is an effective mechanism for allowing some parts of an interface to be considered more "important" than others without forcing the designer to embed such information in the functional specification itself.
2. To enable the efficient computation of the admissible heuristic on which the optimization algorithm relies (Table 1, line 2), we require that the cost function be factorable as a sum of costs over all elements in the functional specification. That way, the cost of already assigned variables can be computed exactly, and for the remaining variables, the cost of the best feasible widget (i.e., one with smallest cost and which has not been pruned by constraint propagation) is used.
3. To support personalization, the cost function should be parametrized in such a way that the appropriate choice of parameters can result in different styles of user interfaces being favored over others.

We start by defining **\$** to be of the form:

$$\$(R(\mathcal{S}_f), \mathcal{T}) \equiv \sum_{T \in \mathcal{T}} \sum_{i=1}^{|T|-1} (N(R, e_{i-1}, e_i) + \mathcal{M}(R(e_i))) \tag{1}$$

where N is an estimate of the effort of navigating between widgets corresponding to the subsequent interface elements, $e_k \in \mathcal{S}_f$, referenced in a trail, T, and $\mathcal{M}$ is a manipulation cost function that measures how good each widget is for manipulating state variables of a given type. Hence, the cost of a rendering is the sum of the costs of each user operation recorded in the trace.

Equation 1 satisfies the first of the three requirements, but requires re-analyzing the entire user trace each time a new cost estimate is necessary, and it fails to satisfy the remaining two requirements.

To address those limitations, we first define $\mathcal{N} : \{\mathsf{sw}, \mathsf{lv}, \mathsf{ent}\} \times \mathcal{W}_c \mapsto \Re$ to be a function, specific to container widgets, that reflects the cost associated with navigating through a rendered interface. In particular, there are three ways (denoted sw, lv, and ent) in which users can transition through container widgets (Figure 8). If we consider a container widget w representing an interface element e, the three transitions are: *entering* (ent), when a descendant

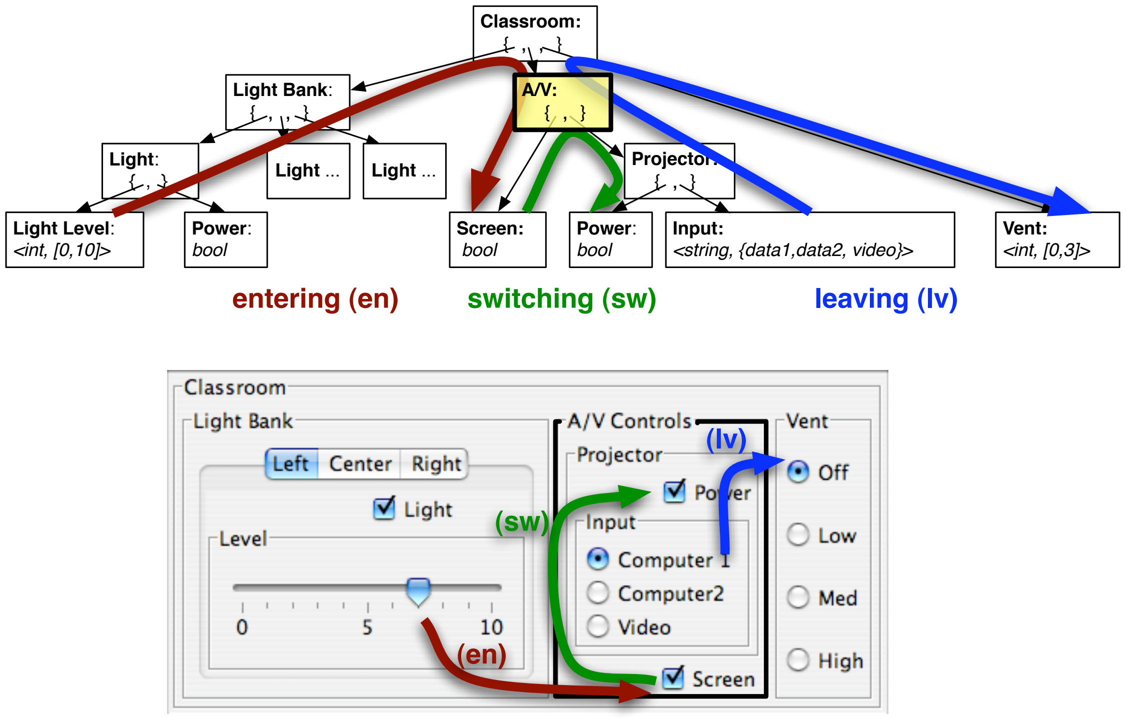


Figure 8: Three types of transitions between elements of a user interface illustrated with respect to the A/V interior node: *entering* (ent), when a descendant of A/V node is manipulated following an element that is not its descendant; *sibling switching* (sw), when user manipulates two different descendants of the A/V node; and *leaving* (lv), when a user manipulates a descendant of A/V and then navigates to an element outside of the A/V sub-tree.

of e is manipulated following an element that is not its descendant; *sibling switching* (sw), when user manipulates two elements that belong to two different descendants of e; and *leaving* (lv), when a user manipulates a descendant of e and then navigates to an element outside of the e sub-tree. For different types of container widgets, these three transitions are predicted to increase user effort in different ways. For example, suppose that e is rendered with a tab pane widget. Then $\mathcal{N}(\mathsf{sw}, e)$, which denotes the cost of switching between its children, would be high, because this maneuver always requires clicking on a tab pane. Leaving a tab widget requires no extra interactions with the tab. Entering a tab pane usually requires extra effort, unless the tab that the user is about to access has been previously selected. In the 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.

Recall that our interface specification is a hierarchy of interface elements. Assuming a rendition where no shortcuts are inserted between sibling branches in the tree describing the interface, one can unambiguously determine the path between any two elements in the interface. We denote the 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 thus choose the navigation cost function, N, from Equation 1 to be of the form:

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

where $\mathrm{child}(e_k, e_{k-1})$ is true if e_{k-1} is a child of e_k. This formula iterates 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 it is considered to be a sibling *switch* between the children of e_k. If e_{k-1} is a grandchild of e_{k+1}, then the path is moving up the interface description hierarchy, and so it *leaves* e_k. Finally, if the path is moving down the hierarchy, then it is *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 during 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 Supple avoids the pathological situations where some of the weights are 0.

Therefore, we may state the component cost of an interface *element*, $R(e)$, as:

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

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

$$\$(R(\mathcal{S}_f), \mathcal{T}) = \sum_{e \in \mathcal{S}_f} \$(R(e), \mathcal{T}) \tag{4}$$

This cost can now be computed incrementally, element-by-element, as the rendering is constructed. Hence, this formulation of the cost function now satisfies the first two requirements listed at the beginning of this section: it incorporates usage traces to emphasize some parts of the user interface over others, and it allows for incremental computation.

To address the third requirement, we introduce *factor functions* $f : \mathcal{W} \times \mathcal{T} \mapsto \Re$. These functions, which take an assignment of a widget to a specification element and a usage trace as inputs, reflect the presence, absence or intensity of some property of the assigned widget. Because they take the usage trace as an input, they also reflect the expected importance of the underlying element. For example, the following factor

$$f_{slider\_for\_number}(R(e), \mathcal{T}) = \#(\mathcal{T}, e) \times \begin{cases} 1 & \textit{if type of } e = \text{number} \\ & \quad \wedge R(e) = \text{slider} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

will return the usage count for the element if it is of number type and is represented by a slider widget. In all other cases, it will return 0. The following equation illustrates a more complex example:

$$\begin{aligned} & f_{list\_undersize}(R(e), \mathcal{T}) = \\ & \#(\mathcal{T}, e) \times \begin{cases} \frac{\text{number of choices}}{\text{list size}} & \textit{if } R(e) = \text{list} \\ & \wedge \text{ number of choices} > \text{list size} \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{6}$$

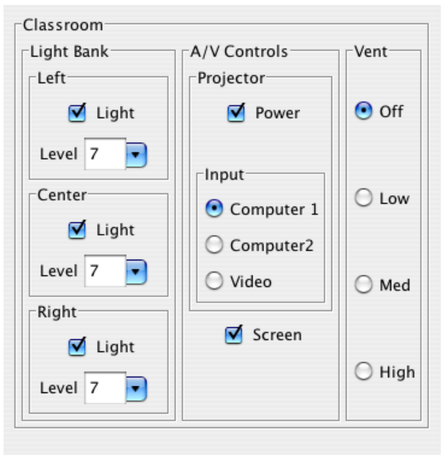
This factor favors larger list widgets in cases where a large number of discrete choices needs to be displayed to the user. The design of this factor was motivated by the fact that the quantity $\frac{\text{number of choices}}{\text{list size}}$ is typically correlated with scrolling performance [35].

The factors can also be used to compute components of the navigation cost $\mathcal{N}$, for example:

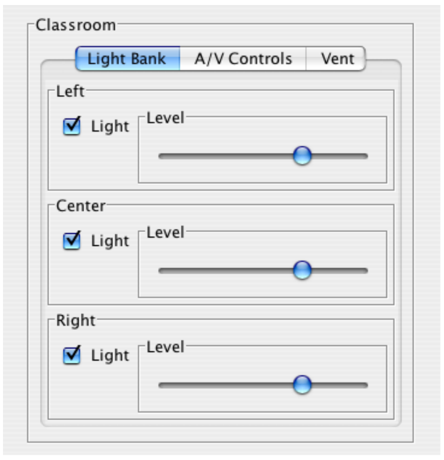
$$f_{tab\_switch}(R(e), \mathcal{T}) = \#_{\mathsf{sw}}(\mathcal{T}, e) \times \begin{cases} 1 & \textit{if } R(e) = \text{tab pane} \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

This factor will return the number of switch transitions for a container element rendered as a tab pane.

By assigning a weight u_k to each factor f_k, and by creating factors for all the foreseeable concerns that might affect perception of interface quality, we can rewrite Equation 3 as follows:



(a)



(b)

Figure 9: Two renderings of the classroom interface both generated under the same size constraint but using different parametrizations of the cost function. Even though both cost functions would cause the same interface to be generated on a larger screen, under this size constraint one emphasizes the ease of navigation (left), while the other favors convenient widgets (right). This demonstrates some of the global concerns that can be captured by our cost function.

$$\$(R(e), \mathcal{T}) = \sum_{k=1}^{K} u_k f_k(R(e), \mathcal{T}) \tag{8}$$

Now the particular style of user interfaces favored by the resulting cost function can be specified by an appropriate choice of weights. This satisfies the last of the three requirements posed for the cost function, namely that it be parametrized to allow for easy personalization of the user interface generation process. Combining Equations 4 and 8, the final formulation of the cost function used by SUPPLE is as follows:

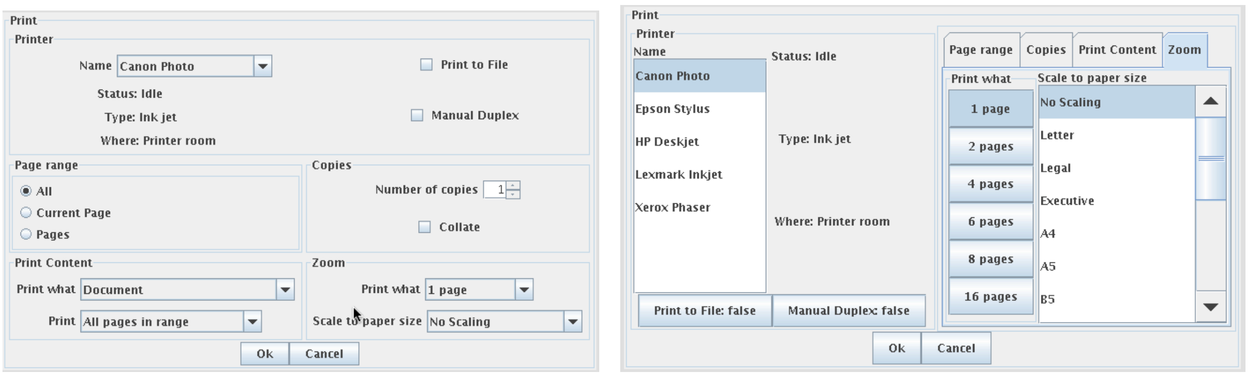
$$\$(R(\mathcal{S}_f), \mathcal{T}) = \sum_{e \in \mathcal{S}_f} \sum_{k=1}^{K} u_k f_k(R(e), \mathcal{T}) \tag{9}$$

This cost function formulation directly captures local layout and widget choices. In combination with screen size constraints, this function also effectively captures certain global trade-offs. For example, Figure 9 shows two renderings of a user interface, both generated under the same size constraint but using different parametrizations of the cost function. Even though both cost functions would cause the same interface to be generated on a larger screen, under this size constraint one emphasizes the ease of navigation (left), while the other favors convenient widgets (right).

Other global concerns cannot be represented using this cost function, but they can be captured with additional interface constraints supplied at the design time (Section 3). For example, the three light controllers in the classroom interface in Figure 9 are constrained to be rendered identically. Such global constraints can be used without sacrificing efficiency as long as these concerns can be propagated efficiently to prune infeasible solutions. An example of a constraint that cannot be incorporated efficiently is a constraint that the dimensions of the complete interface have proportions between 1:1 and 2:3. Such constraint would involve all the variables in the functional specification of the interface and it would rarely be tested before all or almost all variables were assigned. Consequently, given a very large screen, SUPPLE sometimes produces unusually proportioned designs (tall and narrow or short and wide).

The results of our user study suggest, however, that this cost function was expressive enough to capture almost all the design preferences of our participants (Section 8.4.4).

The current implementation of SUPPLE for desktop user interfaces relies on nearly 50 factors. The manual choice of the appropriate weights can be a difficult and error-prone process. For that reason, we have also developed ARNAULD system [22], which automatically learns the right values of these weights based on a small number of preference



Print
Printer
Name: Canon Photo
☐ Print to File
Status: Idle
Type: Ink jet
☐ Manual Duplex
Where: Printer room
Page range
● All
○ Current Page
○ Pages
Copies
Number of copies: 1
☐ Collate
Print Content
Print what: Document
Print: All pages in range
Zoom
Print what: 1 page
Scale to paper size: No Scaling
Ok Cancel

(a)

Print
Printer
Name: Canon Photo, Epson Stylus, HP Deskjet, Lexmark Inkjet, Xerox Phaser
Status: Idle
Type: Ink jet
Where: Printer room
Print to File: false
Manual Duplex: false
Page range | Copies | Print Content | Zoom
Print what: 1 page, 2 pages, 4 pages, 6 pages, 8 pages, 16 pages
Scale to paper size: No Scaling, Letter, Legal, Executive, A4, A5, B5
Ok Cancel

(b)

Figure 10: Two renderings for a print dialog interface automatically generated with different parameterizations of the cost function capturing a user's subjective preferences: (a) a version generated with a cost function designed to produce typical desktop user interfaces, (b) a version generated for touch screen operation.

statements expressed by the user over concrete examples of user interfaces. The results of our subsequent studies [28] indicate that this set of factors is expressive enough to capture most of the *subjective* aesthetic and usability concerns of desktop computer users. As an example, Figure 10 shows two versions of a print dialog interface, one generated with a cost function parametrized to generate typical desktop interfaces, and the other generated for a touch screen operation.

5.2. *Optimizing for Expected Speed of Use*

The previous section described a cost function formulation that is effective for capturing subjective interface design concerns. However, there exist a number of *objective* user interface quality metrics, of which perhaps the most common is the expected time a person would take to perform all input operations required to complete a typical set of tasks with a user interface. This metric was used, for example, as a basis for the Layout Appropriateness measure of interface quality [76].

In this section, we extend our optimization framework to generate user interfaces that are optimized for a user's *performance*, given a predictive model of how fast a person can perform basic user interface operations such as pointing, dragging, list selections and performing multiple clicks.

We start by defining the cost function explicitly as the *Expected Manipulation Time EMT*:

$$\begin{aligned} \$(R(\mathcal{S}_f),\mathcal{T}) &= EMT(R(\mathcal{S}_f),\mathcal{T}) \\ &= EMT_{nav}(R(\mathcal{S}_f),\mathcal{T}) + \textstyle\sum_{e\in\mathcal{S}_f} EMT_{manip}(R(e),\mathcal{T}) \end{aligned} \tag{10}$$

Here, EMT_{nav} is the expected time to navigate the interface (that is, to move from one primitive widget to another, potentially invoking new windows or switching tabs on the way), and EMT_{manip} is the expected time to manipulate a widget (0 for layout widgets). This equation is equivalent to the initial cost function formulation from Equation 1 as long as EMT_{nav} and EMT_{manip} capture all the transition and widget access counts from the trace $\mathcal{T}$. This equation also captures the extent to which expected movement time can be factored: the time to manipulate individual widgets can be computed independently of other parts of the user interface, but the time to navigate the interface cannot be computed until all widgets and layout elements have been chosen. This has implications for the efficiency of the branch-and-bound algorithm, because a substantial portion of the *estimatedSolutionCost* from line 2 in Table 1 cannot be computed until all the variables have been assigned, thus limiting the effectiveness of the admissible heuristic guiding the search.

In the rest of this section, we confront this problem. We begin, however, with a brief description of the process for computing EMT_{manip} for primitive widgets.

5.2.1. *Computing EMT_{manip}*

Many widgets can be operated in more than one way depending on the specific data being controlled and on the user's motor capabilities. For example, a list widget, if it is large enough to show every item, can be operated just by a single click. However, if some of the list elements are occluded, then the user may need to scroll before selecting one of the not-presently-visible elements. Scrolling may be accomplished by dragging the elevator, clicking multiple times on the up/down buttons, depressing an up/down button for a short period of time, or by clicking multiple times in the scrolling region above or below the elevator. Which of these options is fastest depends on how far the user needs to scroll and on how efficiently (if at all) she can perform a drag operation or multiple clicks.

To accommodate the uncertainty about what value the user will select while interacting with a widget, we assign a uniform probability to the possible values that might be selected and then compute the expected manipulation time. To address the choice of ways the widget may be operated (e.g., dragging the elevator versus multiple clicks on a button), Supple computes the EMT_{manip} for each possible method and chooses the minimal value. One cannot decide *a priori* which interaction type is the fastest for a particular widget type because the outcome depends on the circumstances of a particular user (e.g., some eye tracking software does not provide support for dragging).

When computing movement times towards rectangular targets, Supple uses the length of the smaller of the two sides as the target size, as suggested by MacKenzie and Buxton [51]. Although more accurate models for two-dimensional pointing have been developed for typical mouse users [2, 30], those models are unlikely to be equally appropriate for unusual devices, interaction techniques, and users with motor impairments, and we found the approximate approach to be adequate for our purposes.

Finally, note that in order to estimate the movement time *between* widgets, one must take into account the size of the target to be clicked at the end of the movement. That means that the first click on any widget counts toward the navigation time (EMT_{nav}) and not the time to manipulate the widget. Thus the EMT_{manip} for a checkbox, for example, is 0 and the size of the checkbox affects the estimated time to navigate the interface. This increases the urgency of bounding EMT_{nav} before all nodes in the $\mathcal{S}_f$ have been assigned a concrete widget; the next subsection explains how this is done.

5.2.2. *Computing a Lower Bound for EMT_{nav}*

The key to Supple's branch-and-bound search is being able to efficiently bound the cost, including EMT_{nav}, for widgets which have not yet been chosen. Without such a bound, the search took many hours to generate even simple interfaces.

To compute a lower bound on EMT_{nav} that is applicable even when some widgets and layouts have yet to be chosen, we proceed as follows. First, for each unassigned leaf node, e, we compute a rectangular area that is guaranteed to be covered by all of the widgets which are compatible with e; that is, we compute the minimum width of all compatible widgets and separately find the minimum height, as illustrated below.

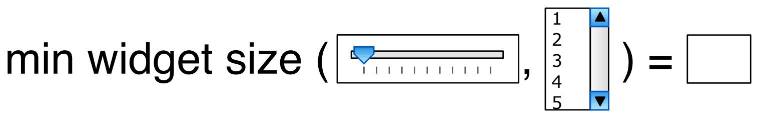


Figure 11: Computing minimum of widget sizes for primitive widgets.

One may now propagate these bounds upwards to compute the minimum sizes for all layout widgets corresponding to interior nodes in the functional specification. For example, the width of an interior node with a horizontal layout is greater than or equal to the sum of the lower bounds of its children's widths. If an interior node has not yet been assigned a specific layout, then we again independently compute the minimum of the possible dimensions.

Note, however, that in this case, for each element contained within a layout element (like the Button A in Figure 12), our estimate also provides the minimum distance from the edges of the layout element to the contained element. As a result, Supple computes the most *compact* possible layout for an interface and thus the shortest possible distance between any pair of elements, as illustrated in Figure 13.

To provide a lower bound on the time to move between elements e_s and e_t, we use the shortest possible distance between the pair and the *largest* possible target size among the set of widgets which are compatible with the target,

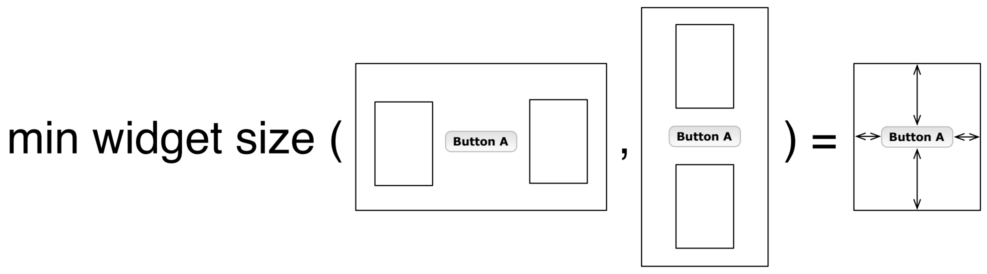


Figure 12: Computing minimum of widget sizes for container widgets. The result is not only the minimum dimensions of the bounding box, but also the minimum distance between any of the enclosed elements and the bounding box.

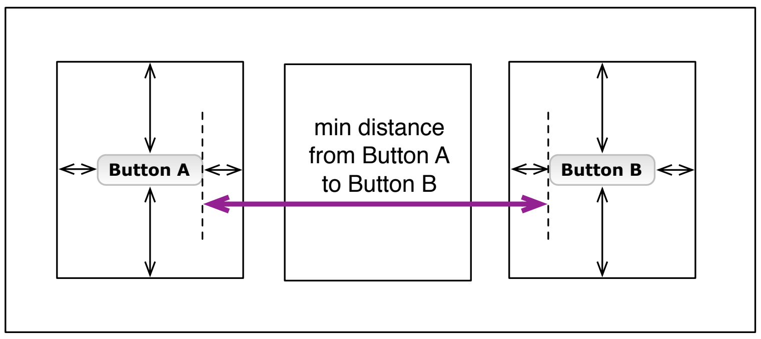


Figure 13: Computing minimum distance between two elements in a layout.

e_t, because movement times grow with the distance and decrease with the size of the target. Supple updates these estimates every time an assignment is made (or undone via backtracking) to any node in the functional specification during the branch-and-bound search process.

More complex layout elements such as tab panes, pop-up panes, or pop-up windows make this process only slightly more complicated; most notably, they require that multiple trajectories are considered if a node on a path between two widgets can be represented by a tab or a pop-up. However, the principle of this approach remains unchanged.

Our results (Section 7.5) show that this lower bound on EMT_{nav} resulted in dramatic improvements to the algorithm performance.

In this section, we have assumed the availability of a model that can predict how long a person would take on average to perform basic user interface operations such as clicking on a distant target, dragging, selecting from a list, or performing multiple clicks on the same object. Fitts' law [15]—a two parameter regression model—and related models (e.g., a related model developed for scrolling performance [35]) are typically used for the purpose. We have previously demonstrated that these approaches poorly capture individual differences among people with unusual abilities or who use atypical devices [27]. We have therefore developed Ability Modeler [27, 28], which automatically selects the features of and then trains a custom regression model for each user. Figure 33 in Section 8 shows several examples of user interfaces generated based on a personalized ability model produced by Ability Modeler.

5.3. *Capturing Consistency Across Interfaces For Different Devices*

Supple enables people to access their applications on a variety of devices. This is a welcome opportunity but also a challenge: users may need to learn several versions of a user interface. To alleviate this problem, newly created user interfaces for any particular application—even if they are created for a novel device—should be *consistent* with previously created ones that the user is already familiar with. Consistency can be achieved at several different levels, such as functionality, vocabulary, appearance, branding, and more [72]. By creating all versions of a user interface from the same model, Supple naturally supports consistency at the level of functionality and vocabulary. In this section, we present an extension to Supple's cost function that allows it to account for dissimilarities in visual appearance and organization between pairs of interfaces. The objective is, if an interface was once rendered on a particular device (for example, a desktop computer) and it now needs to be rendered for a different platform (for example, a PDA), the new interface should strike a balance between being optimally adapted to the new platform and resembling the previous interface.

For that reason, we extended SUPPLE's cost function to include a measure of dissimilarity between the current rendering R and a previous *reference* rendering R_{ref}:

$$\$(R(\mathcal{S}_f), \mathcal{T}, R_{ref}(\mathcal{S}_f)) = \$(R(\mathcal{S}_f), \mathcal{T}) + \alpha_s \Delta(R(\mathcal{S}_f), R_{ref}(\mathcal{S}_f)) \quad (11)$$

Here, $\mathcal{T}$ as before stands for a user trace, $\$(R(\mathcal{S}_f), \mathcal{T})$ is the original cost function, and $\Delta(R(\mathcal{S}_f), R_{ref}(\mathcal{S}_f))$ is a dissimilarity metric between the current rendering R and the reference rendering R_{ref}. The user-tunable parameter α_s controls the trade-off between a design that would be optimal for the current platform and one that would be maximally similar to the previously seen interface.

As with the cost function introduced in Section 5.1, we define the dissimilarity function as a linear combination of K *factors* $f_k : \mathcal{W} \times \mathcal{W} \mapsto \{0, 1\}$, which for any pair of widgets returns 0 or 1 depending on whether or not the two widgets are similar according to a particular criterion. Each factor corresponds to a different criterion. Because dissimilarity factors are defined in terms of differences between individual widgets, overall dissimilarity factors similarly to the cost function from Section 5.1:

$$\Delta(R(\mathcal{S}_f), R_{ref}(\mathcal{S}_f)) = \sum_{e \in \mathcal{S}_f} \sum_{k=1}^{K} u_k f_k(R(e), R_{ref}(e)) \quad (12)$$

Thus the dissimilarity function can be computed incrementally, supporting efficient computation of an effective admissible heuristic.

5.3.1. Relevant Widget Dissimilarity Features

To find the relevant widget features for comparing visual presentations of interface renderings across different platforms, we generated interfaces for several different applications for several different platforms and examined cross-device pairs that appeared most and least similar to one another. These observations resulted in a preliminary set of widget features. Those relevant to primitive widgets (as opposed to the layout and organization elements) are listed below:

Language {toggle, text, position, icon, color, size, angle} — the primary method(s) a widget uses to convey its value; for example, a slider uses the position, a list uses text and position, a checkbox uses toggle.

Domain visibility {full, partial, current value} — some widgets, like sliders, show the entire domain of possible values, while lists and combo boxes are likely to show only a subset of all possible values and spinners only show the current value.

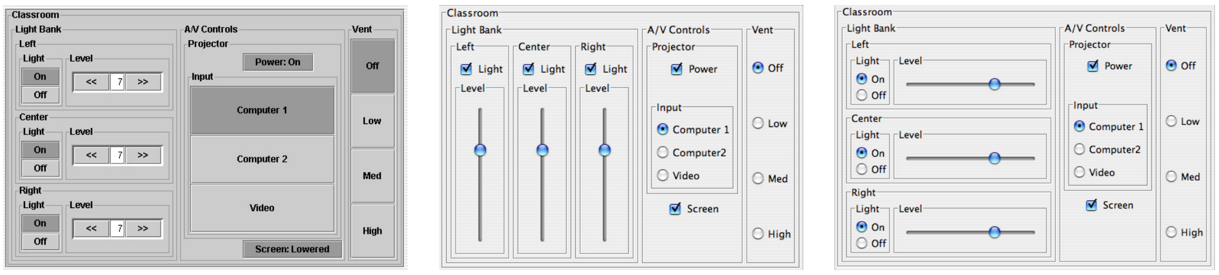
Continuous/discrete — indicates whether or not a widget is capable of changing its value along a continuous range (e.g., a slider can, while a list or a text field are considered discrete).

Variable domain {yes, no} — the domain of possible values can be easily changed at run time for some widgets (e.g., lists), while the set of options is fixed for others (e.g., sets of radio buttons).

Orientation of data presentation {vertical, horizontal, circular} — if the domain of possible values is at least partially visible, there are different ways of arranging these values.

Widget geometry {tall, wide, even} — corresponds to the general appearance of the widget; in some cases it may be different from the orientation of data presentation such as in a short list widget, where elements are arranged vertically but the whole widget may have horizontal (or wide) appearance.

Primary manipulation method {point, drag, text entry} — the primary way of interacting with the widget.



(a) (b) (c)

Figure 14: An illustration of SUPPLE's interface presentation consistency mechanism: (a) a reference touch panel rendering of a classroom controller interface, (b) the rendering SUPPLE considered optimal on a keyboard and pointer device in the absence of similarity information, (c) the rendering SUPPLE produced with the touch panel rendering as a reference.

The features of container widgets (that is, those used to organize other elements) have to do with two salient properties:

Layout geometry {horizontal, grid, vertical} — reflects the basic layout geometry of the enclosed elements.

Impact on visibility {yes, no} — indicates whether or not this widget can affect the visibility of some elements in the user interface; for example, tab panes and pop-up windows can change the visibility of interface elements.

Figure 14 shows an example of a user interface that the user first used on a touch panel, along with two versions of that interface for a desktop computer: one that was generated using only the base cost function and one that included the dissimilarity component.

6. Dynamic Personalization of Automatically Generated UIs

Previous sections demonstrated how to automatically generate user interfaces adapted to a particular device, a person's typical usage pattern, and, possibly, his or her unique motor abilities. However, people's tasks and needs change frequently, and user interfaces adapted to a person's average context may not be ideal in all situations, even though they do capture many of the person's idiosyncrasies. In this section we present two approaches for run-time personalization of SUPPLE-generated user interfaces: system-driven automatic adaptation and user-driven customization.

6.1. System-driven Automatic Adaptation

The inclusion of usage traces in the cost functions allows SUPPLE to generate user interfaces that reflect a person's long-term tasks and usage. However, a person may use the same software for a variety of different types of tasks. Informed by the results of several user studies we conducted [24, 25], we implemented the Split Interface approach [24] in SUPPLE for adapting to the user's task at hand. In Split Interfaces, functionality that is predicted to be immediately useful to the user is copied to a clearly designated adaptive area of the user interface while the main interface remains unchanged. Unlike some other adaptive approaches, Split Interfaces reliably improve both user performance and satisfaction [24].

In contrast to previous implementations of this general approach, which could only adapt contents of menu items [77, 14] or toolbar buttons [24], SUPPLE can adapt *arbitrary* functionality: frequently used but hard to access functionality is copied to the functional specification of the adaptive area and SUPPLE automatically renders it in a manner that is appropriate given the amount of space available in the adaptive part of the interface. For example, if the user frequently changes the print orientation setting, which requires 4 to 6 mouse clicks to access in a typical print dialog box, SUPPLE will automatically copy that functionality to the adaptive part of the main print dialog box (Figure 15).

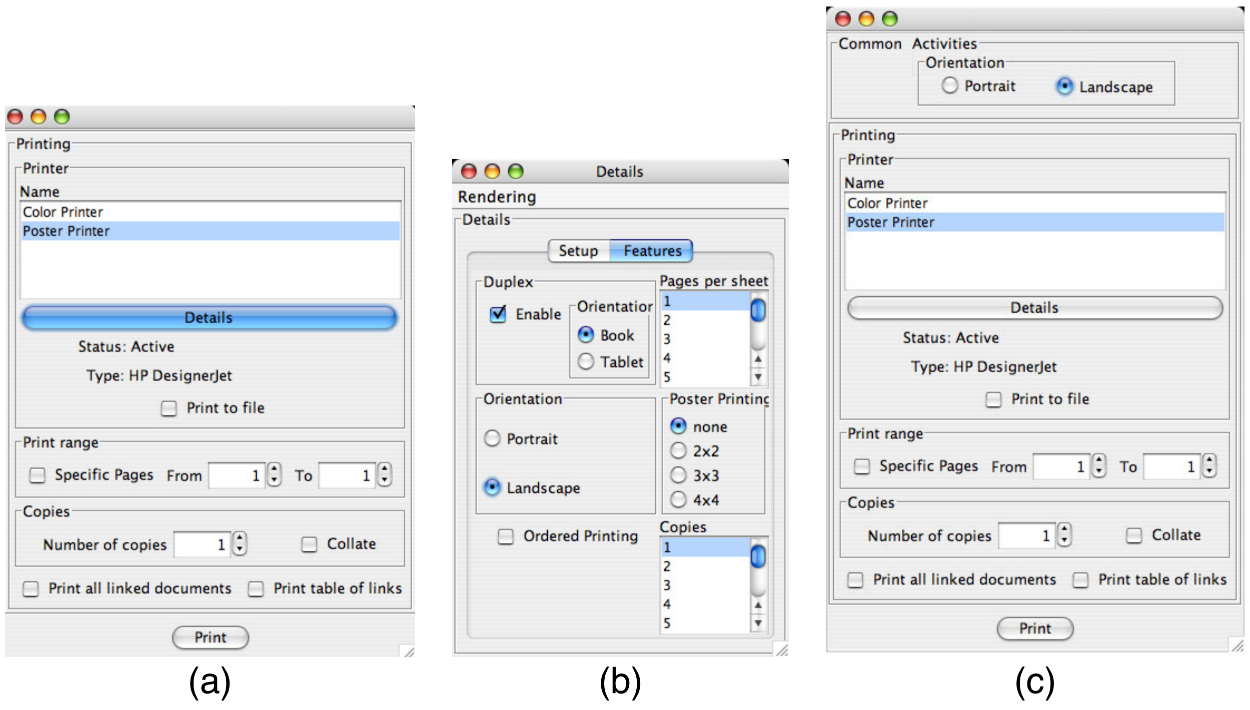


Figure 15: In the original print dialog box, it takes four mouse clicks to select landscape printing: (a) details button, (b) features tab, landscape value and then a click to dismiss the pop-up window. (c) shows the interface after automatical adaptation by Supple, given frequent user manipulation of document orientation; the adapted interface is identical to the one in (a) except for the Common Activities section that is used to render alternative means of accessing frequently used but hard to access functionality from the original interface.

6.2. User-driven Customization

We have already discussed two system-driven approaches to adapting user interfaces in Supple: automatic adaptation to a person's long-term usage patterns by incorporating usage traces in the cost function, and the Split Interface approach for automatic adaptation to the current task. In this section, we introduce a complementary user-driven *customization* mechanism.

Just as with traditional user interfaces, some users may want to customize the organization or presentation of user interfaces generated by Supple. Customization mechanisms offer users control over the user interface and may contribute to significant improvement in satisfaction and performance when used to create custom simplified versions of the interface that are streamlined for the user's individual tasks and habits [52, 53].

Supple includes a comprehensive customization facility that allows a designer or an end user to make explicit changes to an interface, rearranging elements, duplicating functionality, removing elements, and constraining the choice of widgets used to render any part of the functional specification. Operation is simple on a windows and mouse platform: one simply right-clicks the interface element (primitive widget or container), and options for customization are revealed. Duplication and rearrangement are specified with drag-and-drop. This is a much broader range of customizations than those possible with manually-created user interfaces, where presentation customizations are usually restricted to colors and other cosmetic changes, and where organizational changes are typically limited to menus and toolbars.

As illustrated in Figure 16, customizations are recorded as a *customization plan* and they are represented as modifications to the original functional specification rather than as changes to a particular concrete user interface. Specifically, changes to the presence or location of user interface functions are recorded as modifications to the structure of the functional specification while modification to the presentation of the interface (user's choice of a widget or layout for a particular element) are recorded as interface constraints. The interface generation process is thus extended to include an additional pre-processing step, where the customization plan is applied to the functional specification. Only then, the customized functional specification is rendered as a concrete user interface.

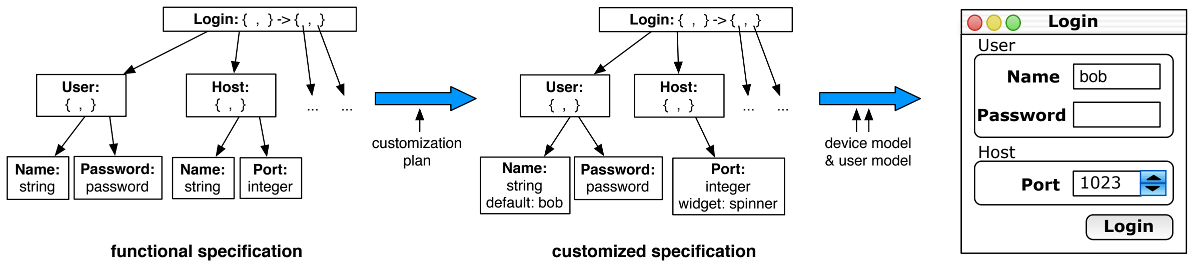


Figure 16: SUPPLE's customization architecture. The user's customization actions are recorded in a *customization plan*. The next time the interface is rendered (possibly in a differently sized window or on a different device) the plan is used to transform the functional specification into a *customized specification* which is then rendered using decision-theoretic optimization as before.

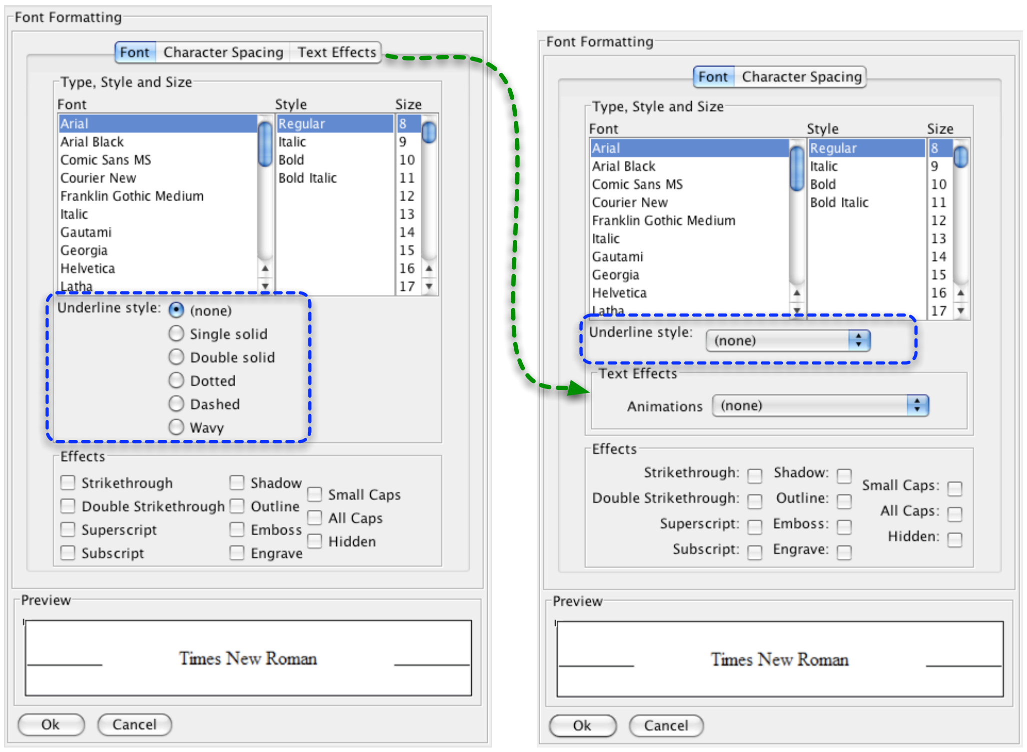


Figure 17: An illustration of the customization mechanism: (left) an interface for a font formatting dialog generated automatically by SUPPLE; (right) the same interface customized by the user: the Text Effects section was moved from a separate tab to the section with main font properties, and the presentation of Underlying Style element was changed from radio buttons to a combo box.

This approach allows customizations performed on one device to be reproduced on other devices, except in cases where equivalent widgets or layouts are not available on the novel device.

Customization plans are editable by users, who may choose to undo any of the earlier customizations, and they can do so even out of order (unlike the typical stack-based implementations of undo functionality). If any of the later customizations depend on the earlier customization the user is attempting to undo, SUPPLE will notify the user of the dependency thereby allowing her to either abandon the undo operation or undo all dependent customizations as well.

The separation of customization plans from the actual interface representation, together with the ability to edit those plans, offers the potential for users to share and collaborate on their user interface modifications.

Figure 17 shows an example of a user interface where both presentation and organization of the interface have been customized. Another more in-depth example is discussed in Section 7.3.

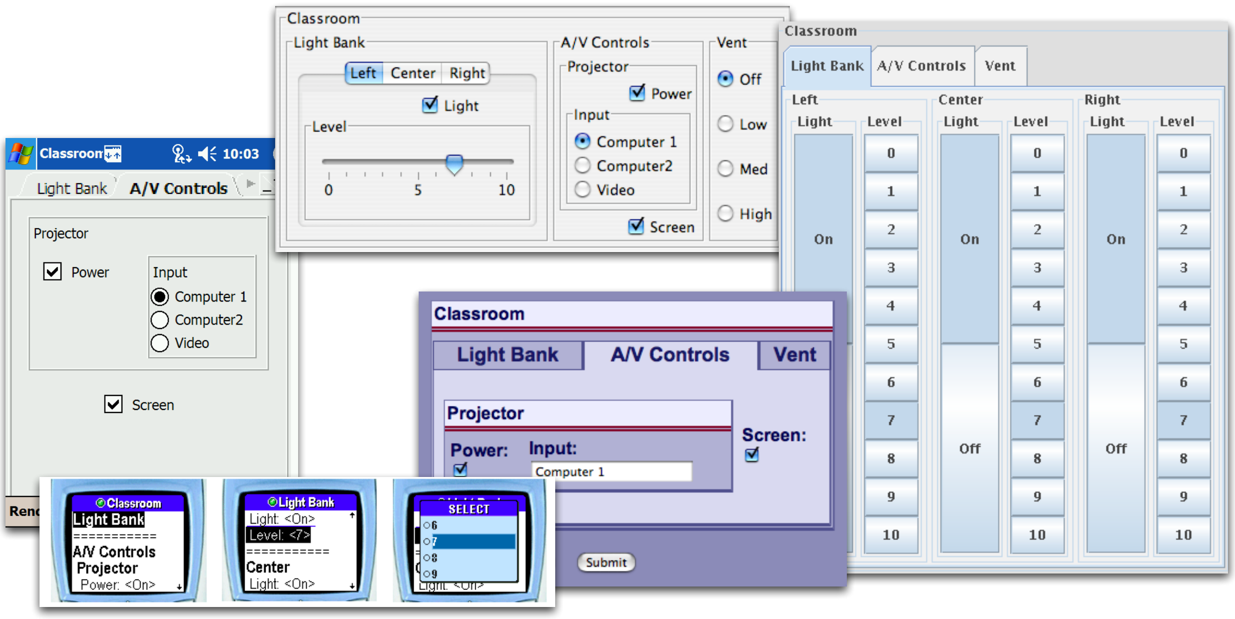


Figure 18: An interface for the classroom controller application rendered automatically by Supple for a PDA, a desktop computer, a touch screen, and HTML browser, and a WAP phone.

7. Evaluation

In this section we examine Supple's technical capabilities and limitations.

7.1. Versatility

We demonstrate Supple's versatility by exhibiting the range of different types of interfaces it has generated. Earlier in this paper, we presented an interactive map-based interface (Figure 2), a fully functional email client (Figure 3), an interface to Amazon web services (Figure 4), an FTP client (Figure 5), an interface for controlling lighting, ventilation and audio-visual equipment in a classroom (Figure 14), and two different print dialog windows (Figures 10 and 15). In this section, Figure 17 provides an example of Supple's customization capabilities on a dialog box for font formatting. Figure 18 illustrates a range of supported devices: the interface for controlling classroom equipment was rendered for such diverse platforms as a touch panel, an HTML browser, a PDA, a desktop computer and a WAP cell phone. Figure 19 shows a user interface for controlling a stereo rendered on a PDA and on a desktop computer. Figure 23 shows a Supple reimplementation of Microsoft's Ribbon interface for Word 2007. Finally, Figure 33 in the next section, shows a font formatting dialog generated for users with different motor abilities.

These examples demonstrate a range of different types of interfaces: device control (classroom and stereo), dialog boxes (font formatting), media-based (map), and data-oriented applications (email and the Amazon client).

Additionally, compared to previous rule-based approaches, optimization robustly and flexibly handles tradeoffs and interactions between choices in different parts of the interface. For example, a rule-based system will likely fail to exploit an increase in screen size (or decrease in interface complexity) by using more convenient but larger widgets. In contrast, Supple's search algorithm always selects an interface that is optimal (with respect to the cost function) for a given interface and device specification. Figure 20 illustrates how Supple robustly degrades the quality of the generated user interfaces as it is presented with devices with progressively narrower screens.

7.2. Adapting To Long-Term Usage Patterns

Both formulations of the cost function described in this paper incorporate usage statistics from a usage model. These statistics impact how Supple generates user interfaces. For example, Figure 21 shows two versions of the classroom interface rendered under the same size constraint. The two interfaces were generated in response to two

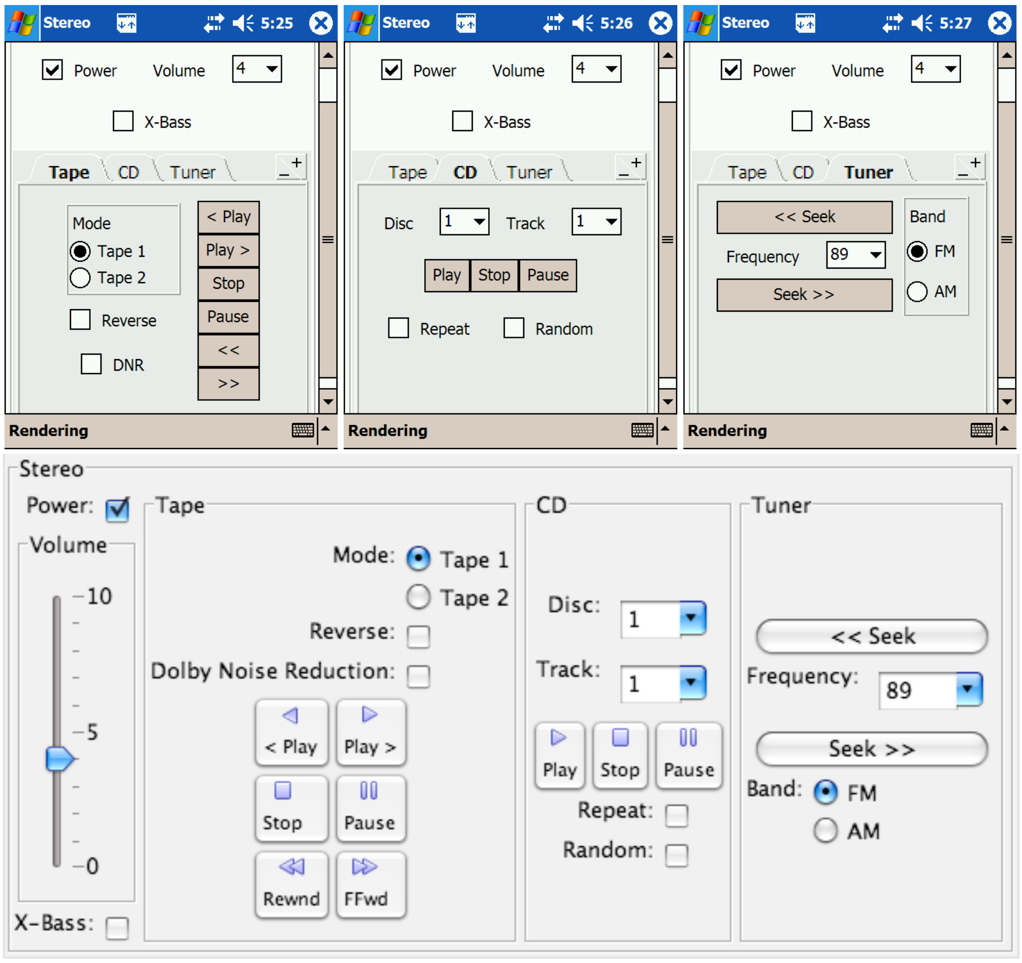


Figure 19: A user interface for controlling a stereo rendered automatically by SUPPLE for a PDA (top) and a desktop computer (bottom).

different usage models. The rendition in Figure 21a was based on a usage trace that represented uniform usage of all the features, while the one in Figure 21b was generated in response to a usage pattern where the three light controls were always manipulated in sequence. The second interface, even though it uses less convenient widgets, makes it easier to navigate between individual light controls than the first one.

7.3. User-driven Customization

Microsoft Ribbon (Figure 22) is an interface innovation introduced in Microsoft Office 2007 as a replacement for menus and toolbars. One of its important properties is that the presentation of the contents of the Ribbon can be adapted based on the width of the document window. The adaptation is performed in several ways, including removing text labels from buttons, re–laying out some of the elements and replacing sections of the Ribbon with pop-up windows. Figure 23a shows a fragment of the Ribbon re-implemented in SUPPLE, while Figure 23b shows that same fragment adapted to fit in a narrower window.

The size adaptation of the Microsoft Ribbon is not automatic—versions for different window widths were designed by hand. An unfortunate consequence of this approach is that no manual customization of the Ribbon is possible: unlike the toolbars used in earlier versions of MS Office, the Ribbon has no mechanism to enable moving, copying, adding, or deleting buttons, panels or other interface elements.

SUPPLE's automatic interface generation algorithm, which takes size as one of the input constraints, automatically provides the size adaptations (Figure 23b). More importantly, however, SUPPLE's customization mechanisms allow

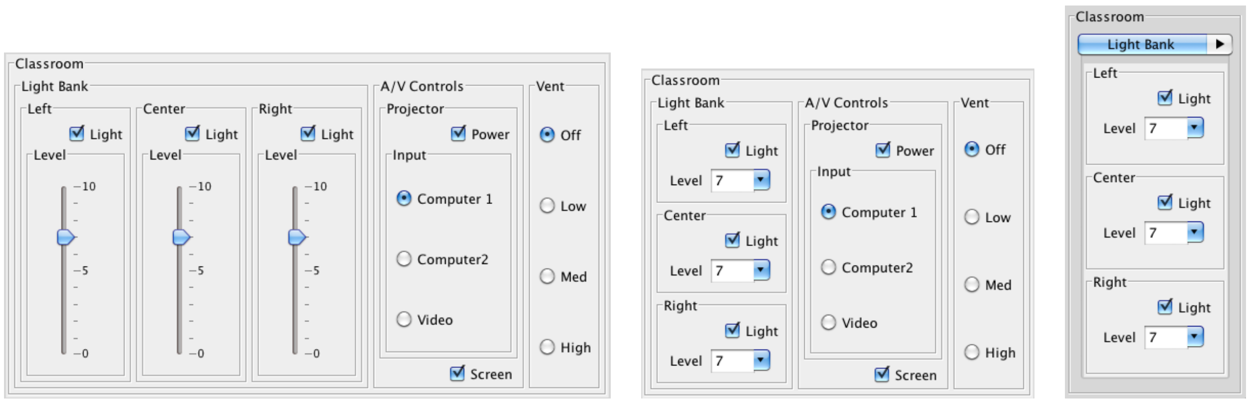


Figure 20: SUPPLE optimally uses the available space and robustly degrades the quality of the rendered interface if presented with a device with a smaller screen size. This figure shows three renderings of a classroom controller on three devices with progressively narrower screens.

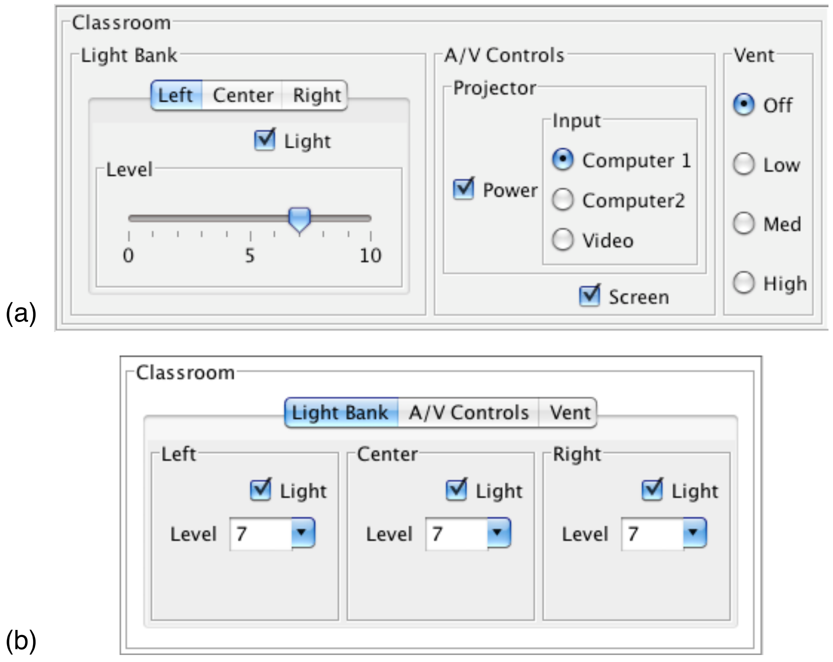


Figure 21: The classroom interface rendered for a small screen size: (a) with an empty user trace (b) with a trace reflecting frequent transitions between individual light controls.

people to add new panels to the SUPPLE version of the Ribbon as well as to move, copy, and delete functionality. The customized Ribbon can be naturally adapted to different size constraints by SUPPLE (Figure 23c). In this case, automatically generated and adapted interactions can improve users' sense of control compared to the manually created solution.

7.4. System Performance

We now systematically evaluate the performance of SUPPLE's optimization algorithm on a variety of user interfaces and for a range of screen size constraints.

The computational problem that SUPPLE solves to generate user interfaces is that of constrained combinatorial optimization. This is a computationally hard problem—exponential in the number of specification elements, in the worst-case—but in practice, most instances of such problems are tractable, with just a small number of instances being

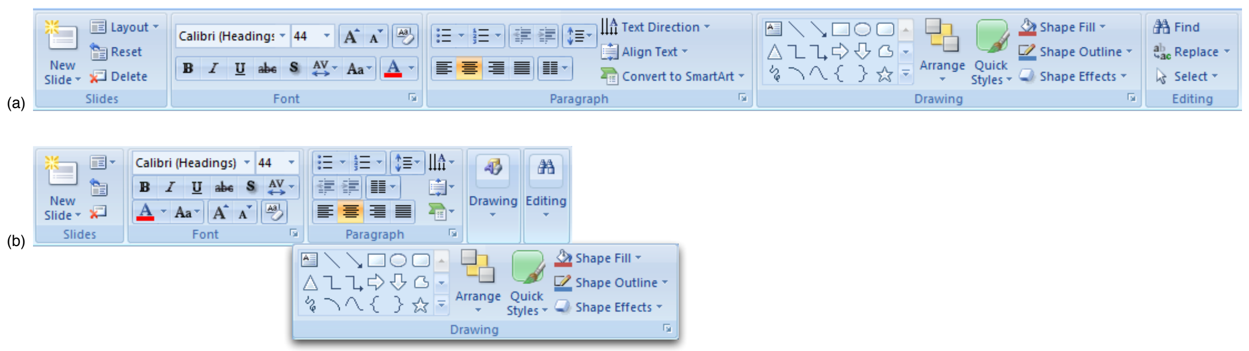


Figure 22: A fragment of the official Microsoft Ribbon (a) presented in a wide window; (b) the same Ribbon fragment adapted to a narrower window: some functionality is now contained in pop-up windows.

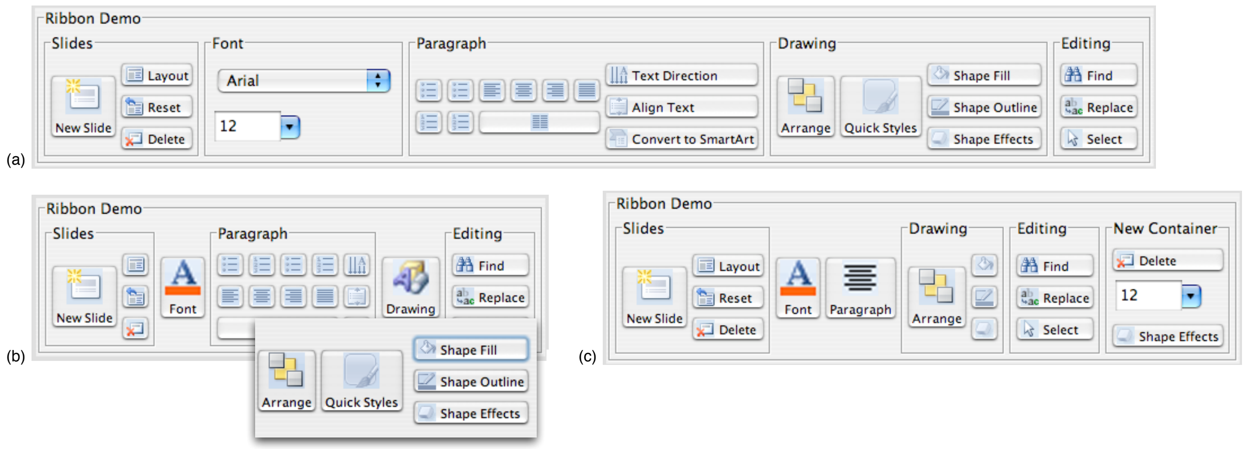


Figure 23: A fragment of our SUPPLE re-implementation of the Microsoft Ribbon (a) rendered for a wide window; (b) SUPPLE automatically provides the size adaptations (enlarging commonly-used functionality based on the user trace), which are manually designed in the original version of the MS Ribbon; (c) unlike the manually designed Ribbon, the SUPPLE version allows users to add, delete, copy and move functionality; in this example, New Container section was added, its contents copied via drag-and-drop operations from other parts of the interface and the Quick Style button was removed from the Drawing panel; the customized SUPPLE version of the Ribbon can still adapt to different size constraints.

substantially harder to solve. Intuitively, given a large amount of screen space, a large fraction of possible renderings will satisfy the size constraints, and the greedy approach of always trying the best widgets first will likely result in quick computation of the optimal interface. Conversely, given a very small amount of screen space, there will be very few or no legal renderings and the constraint propagation process will easily narrow down the solution space to a very small fraction of the original. The hardest problems are therefore somewhere in the middle, in the area where the problem transitions from being under-constrained to being over-constrained. When the existence and the location of these hardest problems are independent of the particular algorithm used, it is frequently referred to as the *phase transition* phenomenon [70, 33, 32]. For some problem spaces, the existence and the location of such phase transitions can be predicted analytically [89]. The space of user interface generation problems, however, is highly discontinuous and therefore hard to investigate analytically. We, therefore, proceed with an empirical investigation.

7.4.1. Variable Ordering Heuristics and the Parallel Algorithm

We empirically investigate both the average and the worst-case performance of SUPPLE's algorithm, using the factored version of the cost function described in Section 5. We start by investigating the properties of the three variable ordering heuristics considered in Section 4: bottom-up, top-down, and minimum remaining values (MRV). To examine a representative cross-section of the problem space for each interface considered, we pick two extreme screen size constraints: one so large that a greedy approach to generating a user interface will succeed, the second just

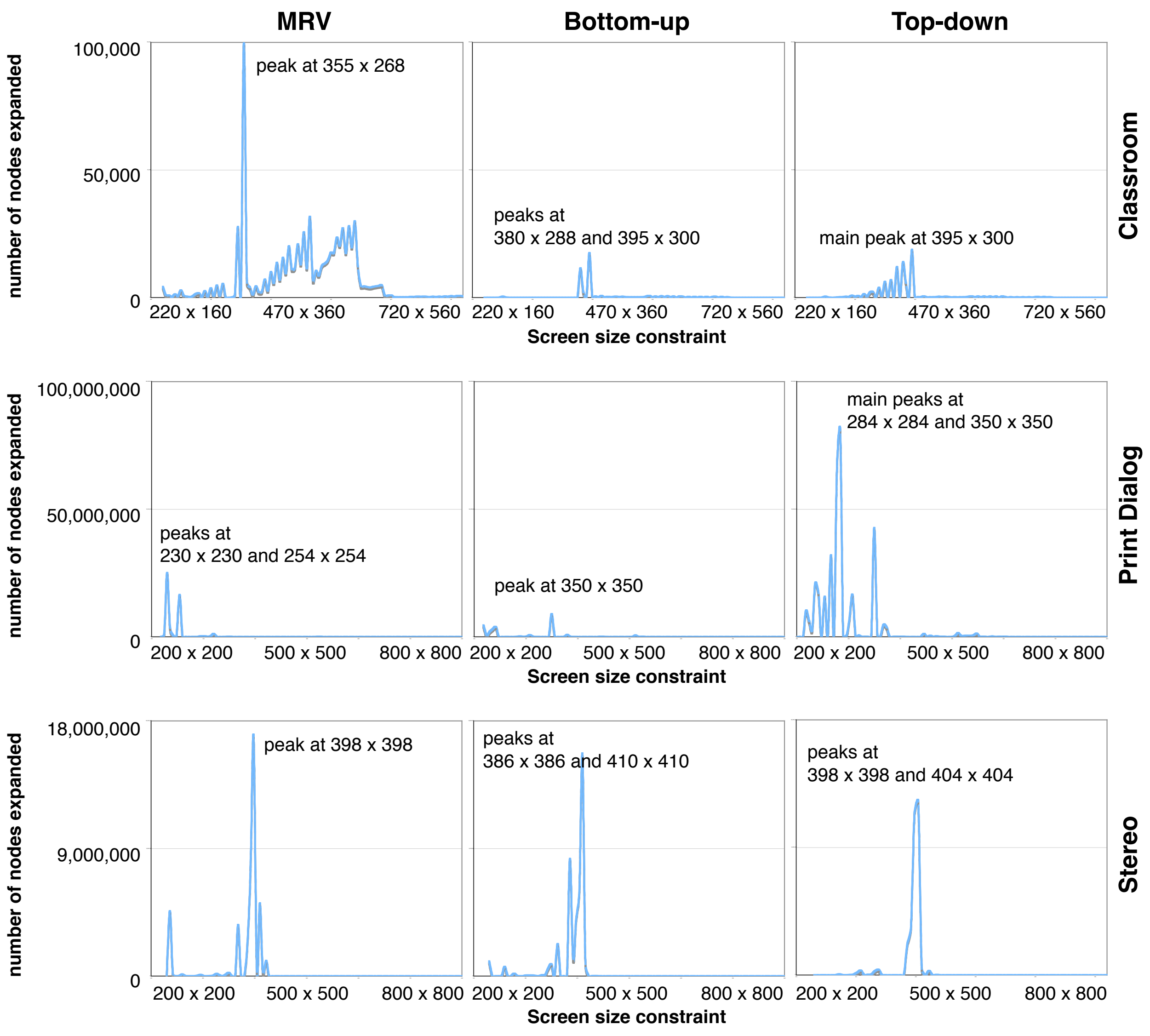


Figure 24: Algorithm performance (quantified by the number of nodes considered by the search algorithm) for three different user interfaces studied systematically over 101 size constraints for three different variable ordering heuristics: minimum remaining values (MRV), bottom-up and top-down.

small enough that no interface can be generated for it. We interpolate at 100 intervals between these two extremes for a total of 101 screen sizes, and for each size we run the optimization algorithm, collecting the following measures:

- The number of nodes expanded by the search algorithm before it finds the *first* solution and before it finds the *best* solution.
- The time taken before the algorithm finds the first solution and before it finds the best solution.

Because execution time is proportional to the number of nodes expanded (see Figure 26) and is hardware-dependent, we omit this measure when comparing different algorithm variants, but we report it in the next subsection when we consider the scalability of the approach.

Interface	Number of unconstrained elements in the specification	Number of possible concrete interfaces	Number of nodes explored				Time taken (seconds)			
			median		maximum		median		maximum	
			Parallel-2	Parallel-3	Parallel-2	Parallel-3	Parallel-2	Parallel-3	Parallel-2	Parallel-3
Map	8	2.16E+02	17	13	23	29	0.07	0.10	0.14	0.35
Test interface A	4	3.51E+02	15	15	106	69	0.13	0.19	0.78	1.53
Test Interface B	8	2.84E+04	40	31	458	269	0.05	0.08	0.84	0.62
Email	25	7.74E+05	49	46	464	387	0.03	0.05	0.34	0.35
Classroom	11	7.80E+07	84	105	2,131	2,125	0.04	0.12	0.52	1.02
Test Interface C	16	1.87E+08	210	50	7,210	6,571	0.14	0.10	2.67	3.81
Ribbon	32	5.44E+08	1,252	1,237	21,757	21,759	0.32	0.42	3.10	4.51
Synthetic	15	1.27E+11	72	40	1,129	836	0.05	0.06	0.44	0.68
Print Dialog	27	3.54E+13	2,024	2,095	120,710	99,035	0.59	0.91	30.31	27.87
Font Formatting	27	2.76E+15	1,025	1,224	106,979	126,135	0.36	0.65	23.17	35.09
Stereo	28	2.79E+17	42	139	323,049	230,900	0.03	0.14	66.15	89.04

Table 2: The performance of the Supple's rendering algorithms with respect to the complexity of the user interface. Both the average case (median) and worst-case (maximum) are reported using time as well as the number of nodes expanded by the search algorithm.

Figure 24 shows the performance of the three variable ordering heuristics across the range of screen size constraints for three interfaces of different levels of complexity: the classroom controller (as the one in Figure 20), a print dialog box (Figure 15), and a stereo controller interface (Figure 19).

This figure illustrates the existence of narrow bands in the problem space where the algorithms perform up to several orders of magnitude worse than in other parts of the problem space. It also illustrates another important phenomenon: the MRV and bottom-up heuristics tend to exhibit their worst performance in slightly different parts of the problem space. This is an important observation because it suggests that actual algorithm-independent phase transition phenomenon may not be taking place, and that combining these two approaches can result in an algorithm that performs orders of magnitude better in the worst-case than either of the two approaches alone.

Motivated by the above results, we implemented two variants of a *parallel algorithm*. The first, which will be referred to as Parallel-2, concurrently runs (in parallel threads) two searches driven by the two variable ordering heuristics whose regions of worst performance do not overlap: the bottom-up and the MRV heuristics. The second, Parallel-3, runs three concurrent searches, one for each of the three heuristics.

In both parallel algorithms, we expected to see the benefit of the individual algorithms experiencing worst performance in different regions of the problem space. In addition, the parallel searches explore the solution space in different orders, coming across solutions at different times, but they share the *bestCost* variable (see Table 1) used for branch and bound pruning. Therefore, we expected that sharing of the cost of the best solution found so far by one of the searches will improve the pruning power of the others.

Figure 25 shows the performance of the two parallel algorithms on the three example interfaces introduced in Figure 24. In each case, the best-performing variant from Figure 24 is also shown for comparison. Note that unlike the previous figure, this one uses a logarithmic scale on the y-axes to highlight large differences in performance.

The average-case performance in all instances remained the same as that of the single search, but, as expected, the worst-case performance improved dramatically: by an order of magnitude in the case of the classroom interface and by *nearly two orders of magnitude* for the print dialog and the stereo interface.

7.4.2. Scalability

Next, we investigate how our approach scales with the complexity of the interfaces it generates.

We evaluated the two parallel algorithms with 11 different user interfaces, a number of which are used as examples throughout this article. In Table 2, we first report for each interface the number of elements in the functional specification (excluding those for which rendering is constrained through the same rendering constraint), and the number of possible interface renderings that the algorithm will consider. As in the previous section, for each interface, we measured the performance at 101 different points throughout the problem space. We measured both the number of nodes explored and the total time taken to produce the best solution. We report both the average case values (the median across all trials) and the worst-case numbers.

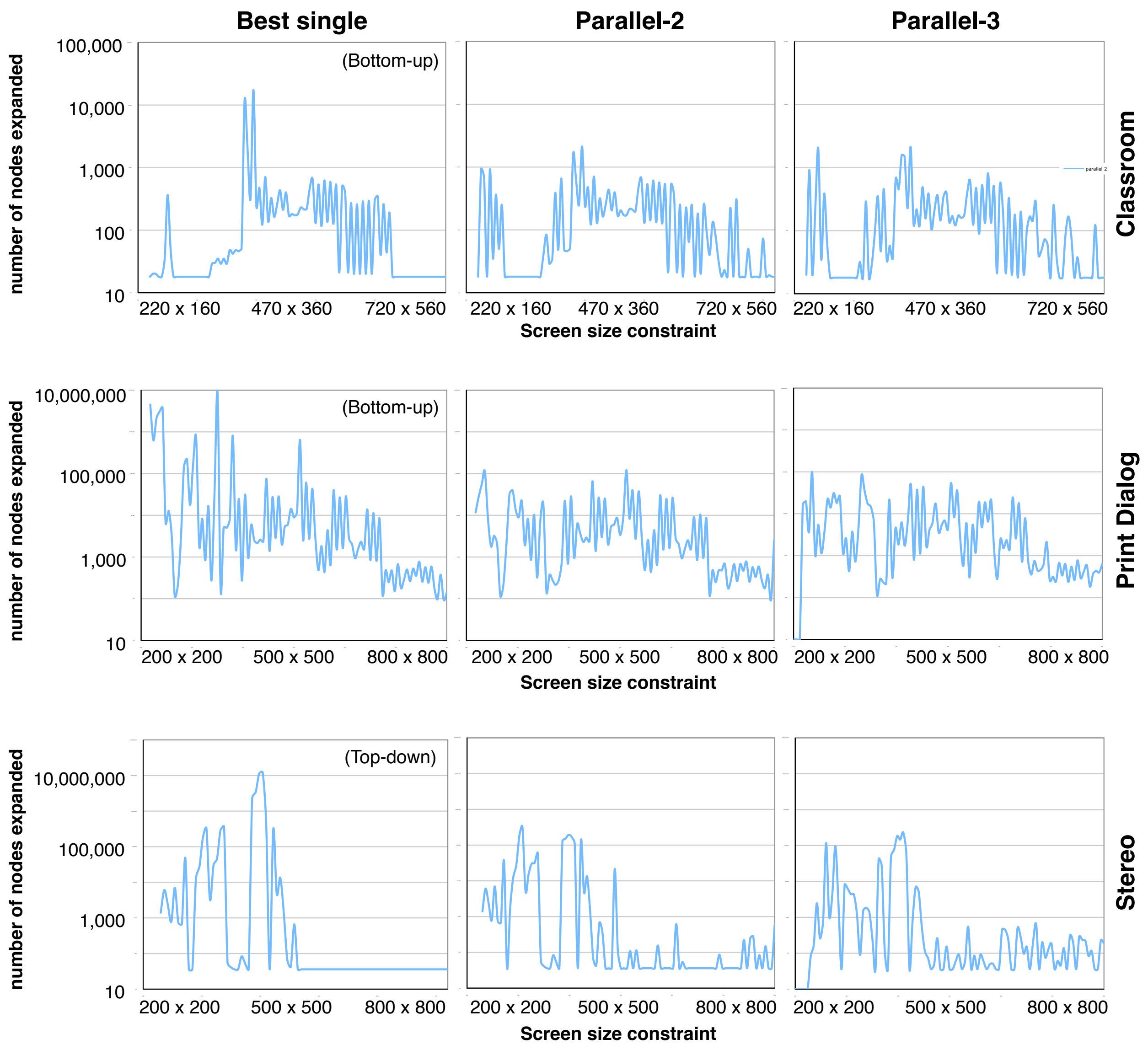


Figure 25: The performance (quantified by the number of nodes considered by the search algorithm) of the two parallel algorithms compared to the best-performing single-threaded variant from Figure 24. The Parallel-2 algorithm combines algorithms driven by the bottom-up and the MRV heuristics. Results are presented for three different user interfaces studied systematically over 101 size constraints. To enable direct comparison, we use a log scale on the y-axis. The worst-case performance of the parallel algorithms is up to two orders of magnitude better than of any of the single algorithms.

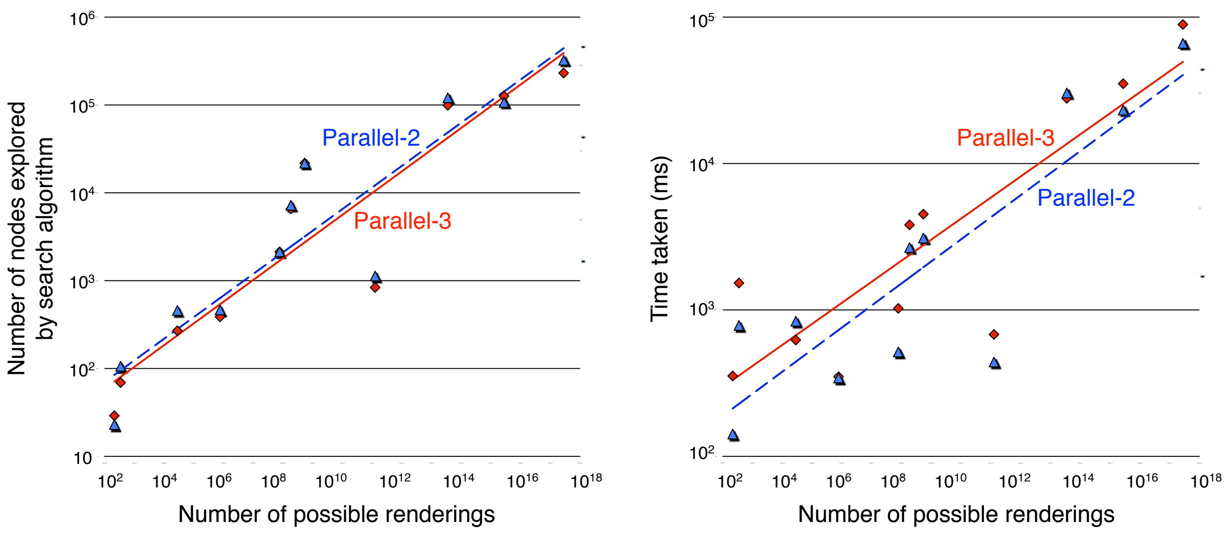


Figure 26: The worst-case performance of the two parallel algorithms with respect to the complexity of the user interface. The x-axes shows the number of possible user interfaces. In the left graph, the y-axis shows the maximum number of nodes explored by the search algorithms before the best solution was found. In the right graph, the y-axis shows the actual time taken. The tests were conducted on a dual core machine. This is why the Parallel-3 algorithm took more time than Parallel-2 even though it explored fewer nodes on average.

Note that the number of possible interfaces considered spans 15 orders of magnitude, from 216 for the map interface to 2.8×10^{17} for the stereo interface. Yet, across this entire range, the median time to find the best interface remains under 1 second.

The median performance of the two algorithms did not vary significantly when measured by the number of nodes explored by the search algorithm. But running on a dual-core processor, the Parallel-2 algorithm was a little faster on average than Parallel-3. In the worst case, the Parallel-3 algorithm typically explored fewer nodes, but required more time. Again, this result reflects the particular hardware architecture used in the experiments.

While the average performance does not correlate with interface complexity, the worst-case performance does. In fact, exponential regression reveals an exponential relationship between the number of possible interfaces and the worst-case execution time or the number of nodes explored by the search algorithm. For the Parallel-2 algorithm, the relationship between the number of possible interfaces, n, and the number of nodes expanded is $22.52 \times n^{0.246}$ ($R^2 = 0.97$), and for Parallel-3 it is $18.83 \times n^{0.247}$ ($R^2 = 0.94$). Figure 26 illustrates these relationships. Of course, because the exponents smaller than 1, the performance scales sub-linearly, specifically as a root of n. Furthermore, the nearly identical exponents suggest that there is not substantial difference in performance between the two parallel algorithms. This is consistent with our earlier observation that the worst-case performance for the top-down variable ordering tended to overlap with with one of the other two (Figure 24).

7.4.3. Importance of Constraint Propagation

Unsurprisingly, constraint propagation has a significant impact on the algorithm's performance. Figure 27 shows the performance of the Parallel-2 algorithm with only forward checking instead of full constraint propagation for the size constraints. For comparison, the dark line toward the bottom of the graph shows the performance of the algorithm with full constraint propagation enabled. For this interface (classroom), the performance was an order of magnitude worse both in the average case (936 versus 84 nodes) and in the worst case (21,088 versus 2,131 nodes). The performance of the algorithm with constraint propagation entirely turned off was too slow to measure.

7.5. Performance When Optimizing For Expected Speed of Use

The cost function introduced in Section 5.2 allows SUPPLE to generate user interfaces that are predicted to be the fastest for a particular person to use. The structure of that cost function does not support as efficient computation of an admissible heuristic for guiding the search as does the first cost function that was used for earlier analyses.

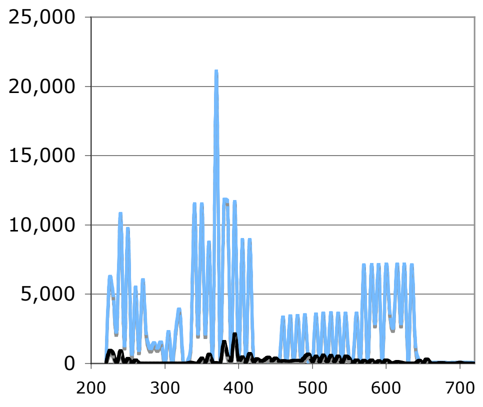


Figure 27: The worst-case performance of the parallel-2 algorithm with only forward checking (a limited version of constraint propagation) with respect to the complexity of the user interface. The x-axis shows the number of possible user interfaces and the y-axis shows the maximum number of nodes explored by the search algorithm. For comparison, the performance of the algorithm with full propagation of the size constraints enabled is shown in solid black line (bottom of the graph).

With this cost function, Supple needed between 3.6 seconds and 20.6 minutes to compute user interfaces. These results take advantage of the lower-bound estimation method for EMT_{nav}, which reduced the runtime for one of the less complex interfaces from over 5 hours to 3.6 seconds, and without which more complex interfaces would have required days to be rendered.

We note that execution times on the order of 10-20 minutes (in the worst case) will still allow practical deployment of the system, if caching is used, for users whose conditions do not change frequently.

7.6. Model Complexity

Comparisons of code quantity among different approaches are often controversial. Yet, we feel it is useful to report the amount of code[3] devoted to the description and management of the user interface for several of the examples reported in this paper. These numbers are reported in Table 3. While we do not have the data showing how much code would be required to build analogous interfaces by hand, the numbers in Table 3 provide some evidence that our approach does not impose excessive burden on the programmer.

	Classroom	**Map**	**Email**	**Amazon**	**Font Formatting**	**Stereo**	**Ribbon**
Lines of user interface code	77	70	515	59	125	125	140

Table 3: Lines of code used to construct and manage the user interfaces for several of the applications presented throughout this paper.

8. User Evaluation

In this section we present a user evaluation of a concrete application of Supple: automatically generating user interfaces adapted to the individual abilities of users with motor impairments. As we have argued earlier in the paper, there is a mismatch between the effective abilities of people with motor impairments and what the creators of typical interfaces assume about the user's strength, dexterity, range of motion, and input devices. This mismatch can prevent or impede interaction with computers. In contrast, even users with severe impairments can effectively operate

[3]Numbers were calculated using the Metrics plugin for Eclipse available at `metrics.sourceforge.net` and reflect all method lines in classes devoted to the interface description for each of the examples.

Participant	*Health Condition*	*Device Used*	*Controlled with*
MI01	Spinal degeneration	Mouse	hand
MI02	Cerebral Palsy (CP)	Trackball	chin
MI03	Friedrich's Ataxia	Mouse	hand
MI04	Muscular Dystrophy	Mouse	two hands
MI05	Parkinson's	Mouse	hand
MI06	Spinal Cord Injury	Trackball	backs of the fingers
MI07	Spinal Cord Injury	Trackball	bottom of the wrist
MI08	Undiagnosed; similar to CP	Mouse	fingers
MI09	Spinal Cord Injury	Trackball	bottom of the fist
MI10	Dysgraphia	Mouse	hand
MI11	Spinal Cord Injury	Mouse	hand

Table 4: Detailed information about participants with motor impairments (due to the rarity of some of the conditions, in order to preserve participant anonymity, I report participant genders and ages only in aggregate).

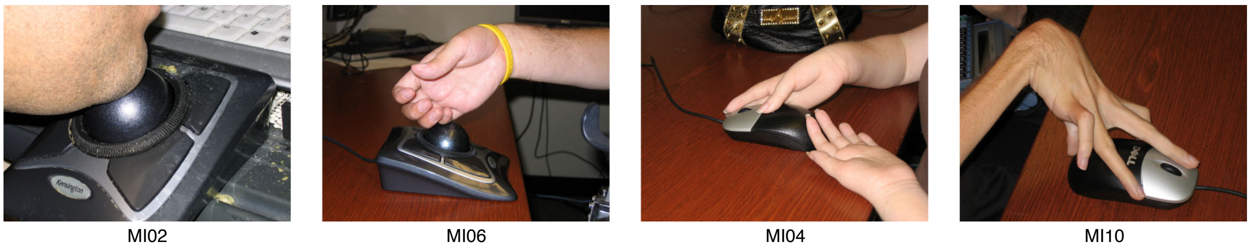


Figure 28: Different strategies employed by our participants to control their pointing devices (MI02 uses his chin).

user interfaces designed with their unique abilities in mind (e.g., [31, 38]). Because of a great variety in individual abilities [5, 39, 41, 46], many such user interfaces are needed. Unlike manual redesign, automatic generation of such individual ability-based interfaces is a scalable solution.

8.1. *Overview of the Approach*

We evaluate two approaches for automatically generating user interfaces adapted to a person's individual motor abilities. The first approach uses the Arnauld system [22] to model users' subjective preferences about what user interfaces are best for them, and it relies on the factored cost function described in Section 5.1 to generate the user interfaces. The second approach uses Ability Modeler [27, 28] to build a model of a person's actual motor abilities; this approach uses the cost function that allows Supple to directly optimize for the expected speed of use (Section 5.2).

We divided the study into two parts, performed on two separate days. During the first part, each participant interacted with Arnauld and then with the Ability Modeler. During the second part, we evaluated participants' performance and satisfaction when using 9 different user interfaces: 3 were baselines copied from existing software, 3 were automatically generated for each participant based on his or her preferences, and 3 were generated based on the participant's measured abilities.

8.2. *Participants*

Altogether, 11 participants with motor impairments (age: 19–56, mean=35; 5 female) and 6 able-bodied participants (age: 21–29, mean=24; 3 female) recruited from the Puget Sound area took part in the study. The abilities of participants with motor impairments spanned a broad range (Table 4), and they used a variety of approaches to control their pointing devices (Figure 28). All but one reported using a computer multiple hours a day and all reported relying on the computer for some critical aspect of their lives.

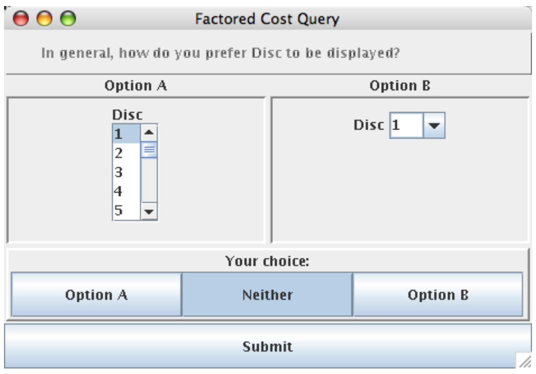


Figure 29: An example of a query used during the active elicitation part of the preference elicitation.

8.3. Apparatus

We used an Apple MacBook Pro (2.33GHz, 3GB RAM) laptop for all parts of the study. Most participants came to our lab for the study and used an external Dell UltraSharp 24” display running at 1920 × 1200 resolution, but 3 of the 11 motor-impaired participants chose to conduct the experiment at an alternative location of their choosing; in these cases, we used the laptop’s built-in 15” display running at the 1440 × 900 resolution.

Each participant had the option of adjusting the parameters of their chosen input device (e.g., tracking speed, button functions). Additionally, we offered the participants with motor impairments the option to use any input device of their choosing, but all of them chose to use either a Dell optical mouse or a Kensington Expert Mouse trackball (Table 4). All able-bodied participants used a mouse. The same equipment with the same settings was used in both parts of the experiment by each participant.

8.4. Part 1: Eliciting Personal Models

8.4.1. Preference Elicitation Tasks

We used Arnauld [22] to elicit participants’ preferences regarding presentation of graphical user interfaces. Arnauld supports two main types of interactions: system-driven *active elicitation* and user-driven *example critiquing*.

During active elicitation participants are presented with queries showing pairs of user interface fragments and asked which, if either, they prefer. The two interface fragments are functionally equivalent, but differ in presentation. The fragments are often as small as a single element, but can be a small subset of an application or an entire application (Figure 29). The queries were generated automatically based on earlier responses from the participant, so each participant saw a different set of queries. The interface fragments used in this study came from two applications: a classroom controller (Figure 20) and a stereo controller (Figure 19). These applications were unrelated to those used in the next phase of this experiment.

During the subsequent example critiquing phase, the participants were shown the interfaces that Supple would generate for them for the classroom and stereo applications. The participants were then offered a chance to suggest improvements to those interfaces. In response, the experimenter would use Supple’s customization capabilities to change the appearance of those interfaces accordingly. These customization actions were used as additional input by Arnauld. If a participant could not offer any suggestions, the experimenter would propose modifications. The original and modified interfaces would then be shown to the participant. Participants’ acceptance or rejection of the modification would be used as further input to Arnauld.

8.4.2. Ability Elicitation Tasks

We used the Ability Modeler [27, 28] to build a model of each participant’s motor abilities. The Ability Modeler builds a predictive model of a person’s motor performance based on the person’s observed performance on four types of basic tasks: pointing, dragging, list selection, and performing multiple clicks on a single target (Figure 30), each repeated multiple times for different target sizes, distances to the target, and the angles of motion (where appropriate). The particular settings used in this study were:

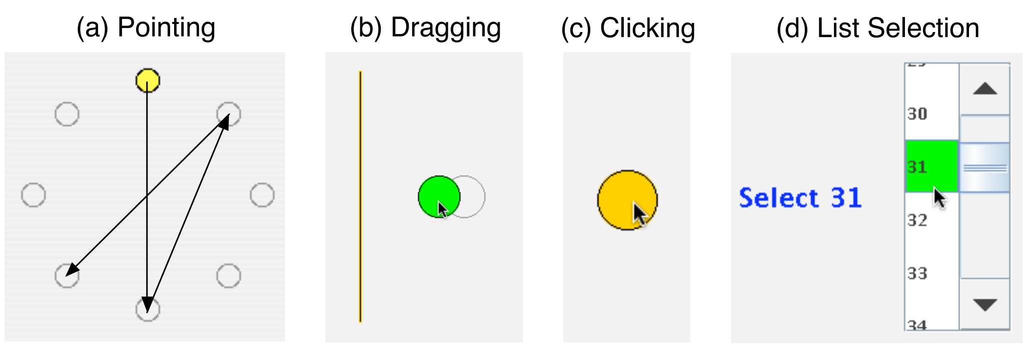


Figure 30: The setup for the performance elicitation study: (a) for pointing tasks; (b) for dragging tasks—here the green dot was constrained to move in only one dimension, simulating the constrained one-dimensional behavior of such draggable widget elements like scroll bar elevators of sliders; (c) for multiple clicks on the same target; (d) for list selection.

- ***Pointing.*** We varied target size (10–90 pixels at 6 discrete levels), distance (25–675 pixels, 7 levels), and movement angle (16 distinct uniformly spaced angles).
- ***Dragging.*** We varied target size (10–40 pixels, 3 levels), distance (100 or 300 pixels) and direction (up, down, left, right).
- ***List Selection.*** We varied the height of the scroll window (5, 10, or 15 items), the distance (measured in the number of items between successive list items to be selected; 10–120, 7 levels), and the minimum size of any clickable element, such as list cells, scroll buttons, scroll bar elevator, or scroll bar width (15, 30, or 60 pixels).
- ***Multiple Clicking.*** We used 5 targets, of diameters varying from 10 to 60 pixels.

8.4.3. Procedure

At the beginning of the session, participants had a chance to adjust input device settings (e.g., tracking speed) and the physical setup (e.g., chair height, monitor position). We then proceeded with preference elicitation followed by ability elicitation, encouraging the participants to rest whenever necessary.

Preference elicitation took 20-30 minutes per participant. Ability elicitation took about 25 minutes for able-bodied participants and between 30 and 90 minutes for motor-impaired participants.

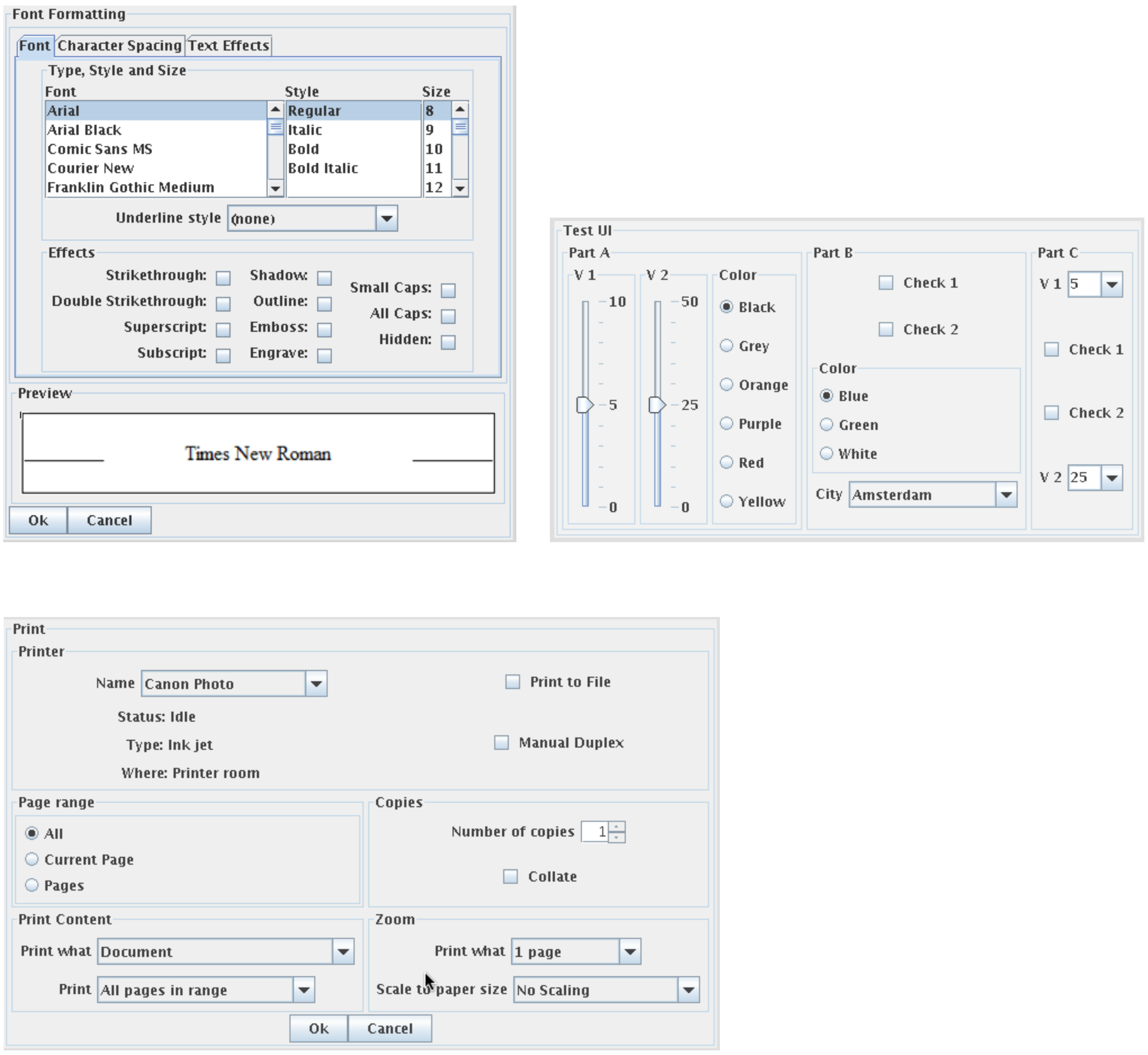
8.4.4. Note on the Validity of Preference Models

Between 30 and 50 active elicitation queries and 5 to 15 example critiquing answers were collected from each participant. Between 51 and 89 preference constraints (mean=64.7) were recorded for each participant. On average, the cost functions generated by Arnauld were consistent with 92.5% of the constraints generated from any one participant's responses. This measure corresponds to a combination of two factors: consistency of participants' responses and the ability of Supple's cost function to capture the nuances of participant's preferences. While this result cannot be used to make conclusions about either the participants or the system alone, it does offer support that the resulting interfaces will reflect users' stated preferences accurately.

8.5. Part 2: Main Experiment

8.5.1. Tasks

We used three different applications for this part of the study: a font formatting dialog box from Microsoft Word 2003, a print dialog box from Microsoft Word 2003, and a synthetic application. The first two applications were chosen because they are frequently used components from popular productivity software. We created the additional synthetic application to include a variety of data types typically found in dialog boxes, some of which were not represented in the two other applications (for example, approximate number selections, which can be represented in an interface with a slider or with discrete selection widgets).


Font Formatting

Font | Character Spacing | Text Effects

Type, Style and Size

Font: Arial, Arial Black, Comic Sans MS, Courier New, Franklin Gothic Medium (selected: Arial)

Style: Regular, Italic, Bold, Bold Italic (selected: Regular)

Size: 8, 9, 10, 11, 12 (selected: 8)

Underline style: (none)

Effects

Strikethrough: ☐ Shadow: ☐ Small Caps: ☐

Double Strikethrough: ☐ Outline: ☐ All Caps: ☐

Superscript: ☐ Emboss: ☐ Hidden: ☐

Subscript: ☐ Engrave: ☐

Preview

Times New Roman

Ok Cancel

Test UI

Part A

V 1: 10, 5, 0 (set to 5)

V 2: 50, 25, 0 (set to 25)

Color: [● Black] ○ Grey ○ Orange ○ Purple ○ Red ○ Yellow

Part B

☐ Check 1

☐ Check 2

Color: [● Blue] ○ Green ○ White

City: Amsterdam

Part C

V 1: 5

☐ Check 1

☐ Check 2

V 2: 25

Print

Printer

Name: Canon Photo

Status: Idle

Type: Ink jet

Where: Printer room

☐ Print to File

☐ Manual Duplex

Page range: [● All] ○ Current Page ○ Pages

Copies

Number of copies: 1

☐ Collate

Print Content

Print what: Document

Print: All pages in range

Zoom

Print what: 1 page

Scale to paper size: No Scaling

Ok Cancel

Figure 31: The baseline variant for the font formatting and print dialog boxes. They were designed to resemble the implementations in MS Office 2003. Two color selection widgets in the font formatting interface were removed, and the preview pane was not functional.

For each application, participants used three distinct interface variants: *baseline*, *preference-based*, and *ability-based*. The baseline interfaces for the font formatting and print dialog boxes were the manufacturer's defaults, re-implemented in SUPPLE to allow for instrumentation, but made to look like the original (see Figure 31). For the synthetic application, we strove for a 'typical' design for a dialog box: it is compact, and relatively uncluttered.

Both the preference- and the ability-based interface variants were automatically generated for each participant individually using the individual preference and ability models that were elicited during the first meeting with the participant.

For the automatically generated user interfaces, we set a space constraint of 750×800 pixels for print and synthetic applications and 850×830 pixels for the font formatting application (see Figures 33 and 34 for examples). These space constraints are larger than the amount of space used by the baseline versions of those applications, but are reasonable for short-lived dialog boxes and our particular hardware configurations. We used the same space constraints for all participants to make results comparable.

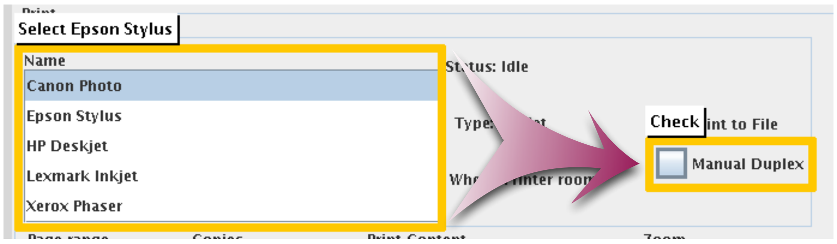


Figure 32: Participants were visually guided to the next element in the interface to be manipulated. The orange border animated smoothly to the next element as soon as the previous task was completed.

Participants performed 6 sets of tasks with each of the interfaces. The first set counted as practice and was not used in the final analysis. Each set included between 9 and 11 operations, such as setting a widget's value or clicking a button; however, if a particular interface included tab panes, interactions with tab panes were recorded as additional operations. For example, if the user had to access Font Style after setting Text Effects in the baseline font formatting interface (Figure 31 top-left), they would have to perform two separate operations: first click on the Font tab and then select the Style.

During each set of tasks, participants were guided visually through the interface by an animated rectangle (Figure 32). An orange border indicated which element was to be manipulated, while the text on the white banner above described the action to be performed. As soon as the participant set the value of a widget or clicked on a tab, the rectangle animated smoothly to the next interface element to indicate the next task to be performed. The animation took 235 ms. We chose to use this approach because we were interested in studying the physical efficiency of the candidate interfaces separate from any other issues that may affect their usability. The animated guide eliminated most of the visual search time required to find the next element, although participants still had to find the right value to select within some widgets.

All tasks were performed entirely with a pointing device without the use of keyboard shortcuts.

8.5.2. *Procedure*

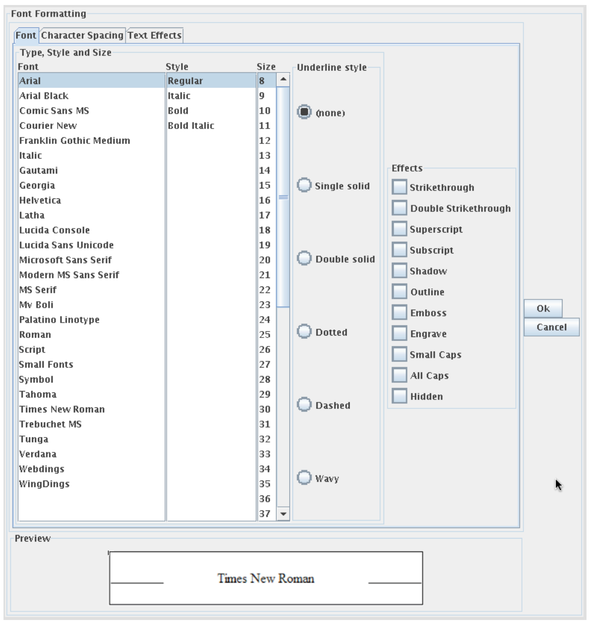
We presented participants with each of the 9 interfaces in turn: 3 applications (font formatting, print dialog, and synthetic) × 3 interface variants (baseline, preference-based, and ability-based). Interface variants belonging to the same application were presented in contiguous groups. With each interface variant, participants performed 6 distinct task sets, the first being considered practice (participants were told to pause and ask clarifying questions during the practice task sets, but to proceed at a consistent pace during the test sets). Participants were encouraged to take a break between task sets.

The tasks performed with each of the 3 interface variants of an application were identical and were presented in the same order. We counterbalanced the order of the interface variants both within each participant and across participants. The order of the applications was counterbalanced across participants.

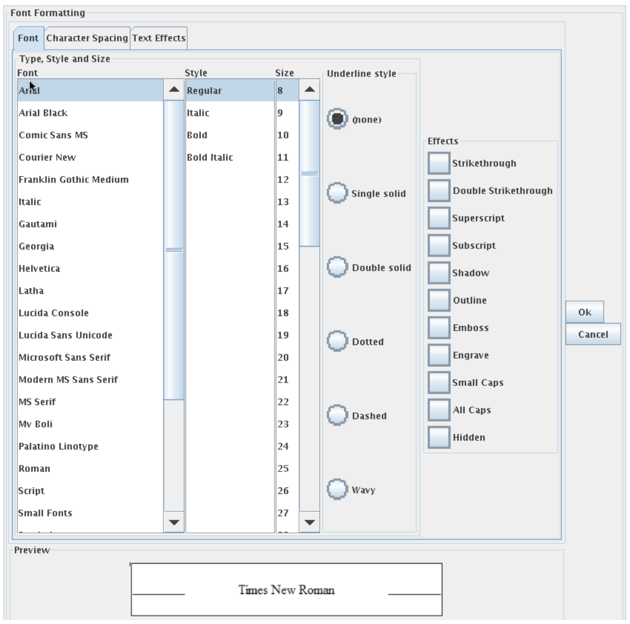
After participants completed testing with each interface variant, we administered a short questionnaire asking them to rate the variant's usability and aesthetics. After each block of three variants (i.e., after each application), we additionally asked participants to rank the three interfaces on efficiency of use and overall preference. Finally, at the end of the study, we administered one more questionnaire recording information about participants' overall computer experience, the computer input devices they typically use, and their impairment (if any).

8.5.3. *Generated Interfaces*

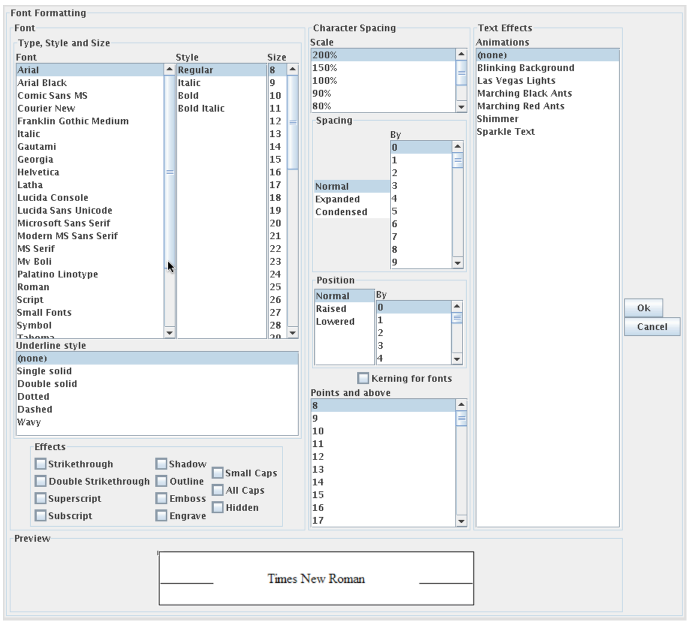
Figure 33 shows three examples of user interfaces generated by Supple based on participants' measured motor capabilities. These "ability-based user interfaces" tended to have widgets with enlarged clickable targets requiring minimal effort to set (e.g., lists and radio buttons instead of combo boxes or spinners). In contrast, user interfaces automatically generated by Supple based on participants' stated preferences (see Figure 34) tended to be very diverse, as each participant had different assumptions about what interfaces would be easier to use for him or her.



AB02



MI02



MI04

Figure 33: User interfaces automatically generated by SUPPLE for the font formatting dialog based on three users' individual motor abilities. The interface generated for AB02 was typical for most able-bodied participants: small targets and tabs allow individual lists to be longer, often eliminating any need for scrolling. MI02 could perform rapid but inaccurate movements; therefore all the interactors in this interface have relatively large targets (at least 30 pixels in each dimension), at the expense of having to perform more scrolling with list widgets. In contrast, MI04 could move mouse slowly but accurately and could use the scroll wheel quickly and accurately; this interface therefore reduces the number of movements necessary by placing all the elements in a single pane, at the expense of using smaller targets and lists that require more scrolling.

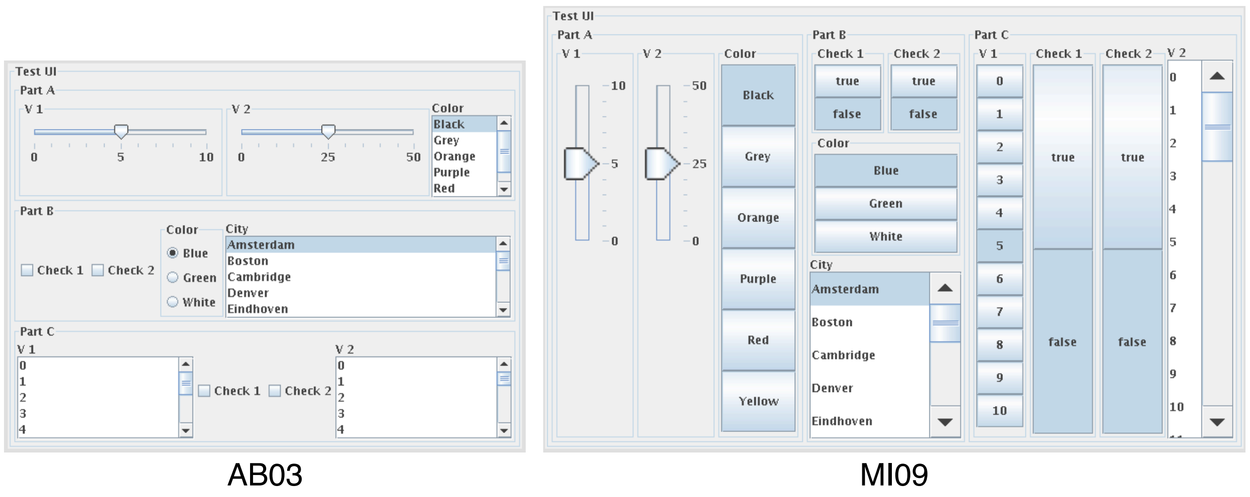


Figure 34: User interfaces for the synthetic application. The baseline interface is shown in comparison to interfaces generated automatically by SUPPLE based on two participants' preferences. Able-bodied participants like AB03 preferred lists to combo boxes, but preferred them to be short; all able-bodied participants also preferred default target sizes to larger ones. As was typical for many participants with motor-impairments, MI09 preferred lists to combo boxes and frequently preferred the lists to reveal a large number of items; MI09 also preferred buttons to either check boxes or radio buttons, and liked larger target sizes.

8.5.4. Design and Analysis

The experiment was a mixed between- and within-subjects factorial design with the following factors and levels:

- *Impairment* {able-bodied (AB), motor-impaired (MI)}
- *Interface variant* {baseline, ability-based, preference-based}
- *Application* {font formatting, print dialog, synthetic}
- *Trial set* {1...5}
- *Participant* {1...17}

Each participants completed $3 \times 3 \times 5 = 45$ trial sets for a total of 765 trial sets (270 for able-bodied and 495 for motor-impaired).

The dependent measures were:

- **Widget manipulation time** captures the time, summed over all operations in a trial set (including errors), spent by the participants manipulating individual widgets. It was measured from the moment of first interaction with a widget (first clicks or mouse wheel scroll in case of lists) to the moment the widget was set to the correct value. For many individual operations involving widgets like buttons, tabs, and lists (if the target element was visible without scrolling), 0 manipulation time resulted, because the initial click was all that was necessary to operate the widget.
- **Interface navigation time** represents the time, summed over all operations in a trial set (including errors), participants spent moving the mouse pointer from one widget to the next; it was measured from the moment of the effective start of the pointer movement to the start of the widget manipulation.
- **Total time** per trial set was calculated as a sum of widget manipulation and interface navigation times.
- **Error rate** per trial set was calculated as the fraction of operations in a set where at least one error was recorded; we regarded "errors" as any clicks that were not part of setting the target widget to the correct value.

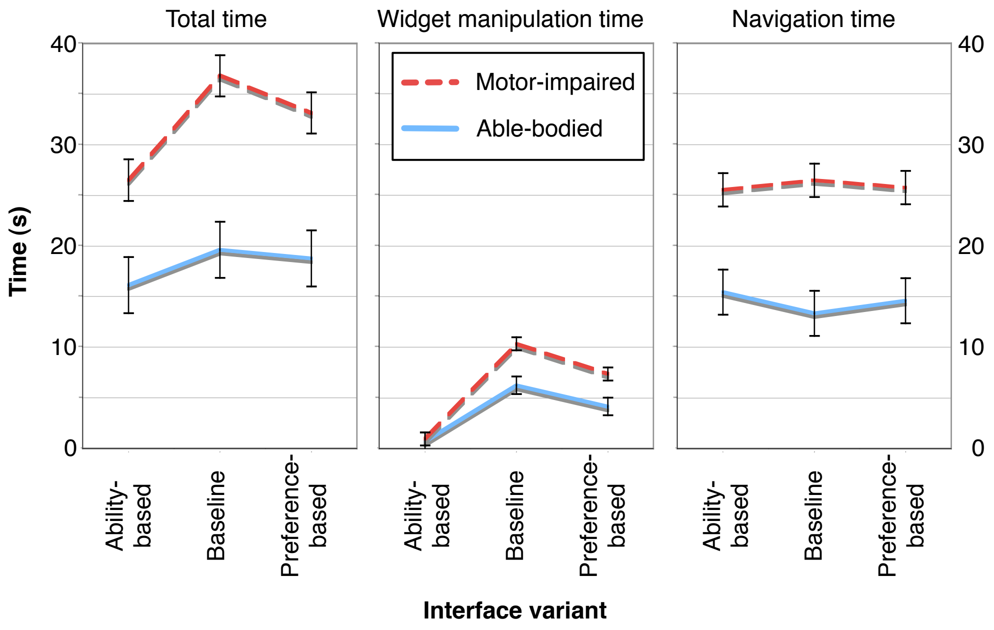


Figure 35: Participant completion times. Both motor-impaired and able-bodied participants were fastest with the ability-based interfaces. The baseline interfaces were slowest to use. Error bars show standard error.

For each application and interface variant combination, we additionally collected 4 subjective measures on a Likert scale (1–7) relating to the interfaces' usability and attractiveness. We also asked the participants to rank-order the 3 interface variants for each application by perceived efficiency and overall preference.

For analysis, we took the logarithm of all timing data to adjust for non-normal distributions, which are often found in such data [4]. We analyzed the timing data using a mixed-effects model analysis of variance with repeated measures: *Impairment*, *Interface variant*, *Application* and *Trial set* were modeled as fixed effects while *Participant* was modeled correctly as a random effect because the levels of this factor were drawn randomly from a larger population. Although such analyses retain larger denominator degrees of freedom, detecting statistical significance is no easier, because wider confidence intervals are used [49, 75]. In these results, we omit reporting the effects of *Application* and *Trial set* because they were not designed to be isomorphic and naturally were expected to result in different performance. As often is the case, the error rate data was highly skewed towards 0 and did not permit analysis of variance. Accordingly, we analyzed error rates as count data, using regression with an exponential distribution [84]. Subjective Likert scale responses were analyzed with ordinal logistic regression [90], and subjective ranking data with the Friedman non-parametric test.

For all measures, additional pairwise comparisons between interface variants were done using a Wilcoxon Signed Rank test with Holm's sequential Bonferroni procedure [37].

8.6. Results

8.6.1. Adjustment of Data

We excluded 2/765 trial sets for two different motor-impaired participants, one due to an error in logging, and one because the participant got distracted for an extended period of time by an unrelated event.

8.6.2. Completion Times

Both *Impairment* ($F_{1,15}$=28.14, $p < .0001$) and *Interface variant* ($F_{2,674}$=228.30, $p < .0001$) had a significant effect on the total task completion time. Motor-impaired users needed on average 32.2s to complete a trial set while able-bodied participants needed only 18.2s. The ability-based interfaces were fastest to use (21.3s), followed by preference-based (26.0s) and baselines (28.2s). A significant interaction between *Impairment* and *Interface variant* ($F_{2,674}$=6.44, $p < .01$) indicates that the two groups saw different gains over the baselines from the two personalized

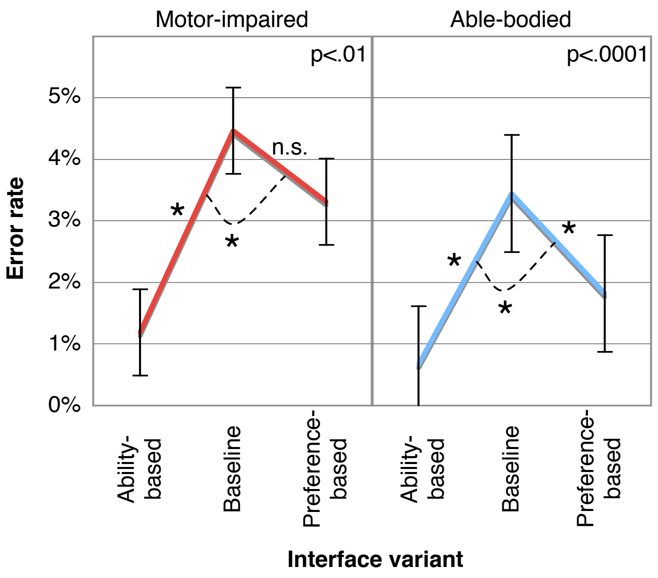


Figure 36: Participant error rates. Both motor-impaired and able-bodied participants made fewest errors with the ability-based interfaces. The baseline interfaces resulted in most errors. Error bars show standard error. Significant pairwise differences are indicated with a star (∗).

interface variants. As illustrated in Figure 35 (left), participants with motor-impairments saw significant gains: a 10% improvement for preference-based and a 28% improvement for ability-based interfaces ($F_{2,438}$=112.17, $p < .0001$). Able-bodied participants saw a relatively smaller, though still significant, benefit of the personalized interfaces: a 4% improvement for preference-based and 18% for ability-based interfaces ($F_{2,220}$=49.36, $p < .0001$).

The differences in performance can be explained by a significant[4] main effect of *Interface variant* on total manipulation time, that is, the time spent actually manipulating the widgets ($\chi^2_{(2,N=763)}$=359, $p < .0001$). With baseline interfaces, participants spent on average 8.29s per trial set manipulating the individual widgets. With preference-based interfaces, this number was 5.76s, while for ability-based interfaces, it was only 0.84s, constituting a nearly 90% reduction compared to baseline interfaces.

For all results reported so far, the pairwise differences between individual interface variants were statistically significant as well.

We additionally observed a significant main effect of *Interface variant* on the total navigation time ($F_{2,674}$=7.76, $p < .001$), explained by the significant difference between baseline and ability-based interfaces ($z = -3180$, $p < .01$). Baseline interfaces required the least amount of navigation time on average (19.9s) while preference- and ability-based interfaces required a little longer to navigate (20.2s and 20.5s, respectively). While statistically significant, these differences were very small — on the order of 3% — and were offset by the much larger differences in total manipulation time. There was a significant interaction between *Impairment* and *Interface variant* with respect to the total navigation time ($F_{2,674}$=9.20, $p < .0001$): for able-bodied participants, navigation time was longer for both of the personalized interfaces ($F_{2,220}$=17.18, $p < .0001$; all pairwise differences were significant as well), while for motor-impaired participants the effect was opposite, though smaller in magnitude and not significant.

8.6.3. Error Rates

There was a significant main effect of *Interface variant* on error rate ($\chi^2_{(5,N=153)}$=55.46, $p < .0001$): while the average error rate for baseline interfaces was 3.96%, it dropped to 2.57% for preference-based interfaces and to 0.93% for ability-based interfaces. This means that participants were both significantly faster *and* more accurate with the ability-based interfaces. There was no significant interaction between *Impairment* and *Interface variant* and

[4]The manipulation time data had bi-modal distribution because for many task sets the total manipulation time was 0. We therefore used a non-parametric Wilcoxon Rank Sum test [88] to analyze these data.

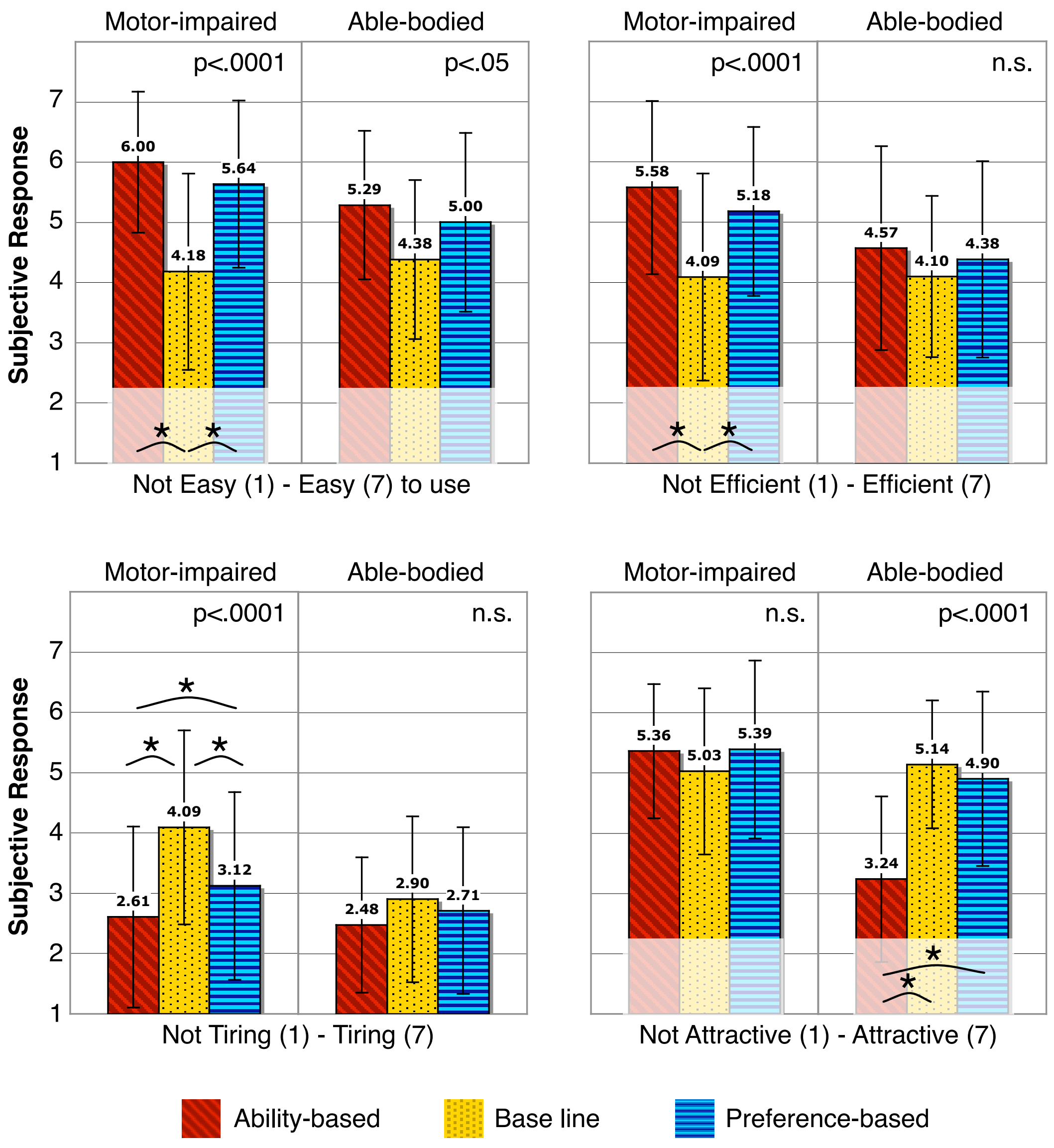


Figure 37: Subjective results. Both groups of participants found ability-based interfaces easiest to use. Motor-impaired participants also felt that they were most efficient and least tiring. Able-bodied participants found ability-based interfaces least attractive but, interestingly, motor-impaired participants saw little difference in attractiveness among the three interface variants. Error bars correspond to standard deviations. Note that on all graphs higher is better except for Not Tiring-Tiring. Significant pairwise differences are indicated with a star ($*$).

the effects were similar and significant for both groups individually ($\chi^2_{(2,N=54)}$=23.66, p < .0001 for able-bodied and $\chi^2_{(2,N=99)}$=11.00, p < .01 for motor-impaired; see Figure 36).

All pairwise differences between individual interface variants for the results reported here are statistically significant, with the exception of the difference between the baseline and preference-based condition for participants with motor impairments.

8.6.4. Subjective Results

On a *Not Easy* (1) – *Easy* (7) scale for ease of use, motor-impaired participants rated ability-based interfaces easiest (6.00), preference-based next (5.64), and baseline most difficult (4.18). Similarly for able-bodied participants: 5.29 for ability-based, 5.00 preference-based and 4.38 for baseline. For both groups, these effects were significant

($\chi^2_{(2,N=99)}$=40.40, $p < .0001$ for motor-impaired, and $\chi^2_{(2,N=63)}$=6.95, $p < .05$ for able-bodied). Additionally, pairwise comparisons showed that participants with motor impairments found both of the automatically generated user interfaces significantly easier to use than the baseline. These subjective results summarized in Figure 37, which also shows all of the statistically significant pairwise comparisons.

On a *Not Efficient* (1) – *Efficient* (7) scale, motor-impaired participants also found ability-based interfaces to be most efficient (5.58), followed by preference-based (5.18) and baseline interfaces (4.09). This effect was significant ($\chi^2_{(2,N=99)}$=23.31, $p < .0001$), but no corresponding significant effect was observed for able-bodied participants. As before, significant pairwise differences only exist between the baseline condition and each of the automatically-generated ones for participants with motor impairments.

Similarly, on a *Not Tiring* (1) – *Tiring* (7) scale for how physically tiring the interfaces were, motor-impaired participants found baseline interfaces to be much more tiring (4.09) than either preference-based (3.12) or ability-based (2.61) variants ($\chi^2_{(2,N=99)}$=25.69, $p < .0001$), while able-bodied participants did not see the three interface variants as significantly different on this scale. All three pairwise differences for this measure were significant for participants with motor impairments.

On a *Not Attractive* (1) – *Attractive* (7) scale for visual presentation, able-bodied participants found ability-based interfaces much less attractive (3.24) than either preference-based (4.90) or baseline variants (5.14). This effect was significant ($\chi^2_{(2,N=63)}$=25.52, $p < .0001$), and so were the pairwise differences between the ability-based and each of the other two conditions. Importantly, motor-impaired participants saw no significant difference in the attractiveness of the different interface variants.

When asked to rank-order the three interface variants of each application by efficiency of use and overall preference (Table 5), both groups of participants ranked ability-based interfaces as most efficient, followed by preference-based, and then baseline interfaces. This result was only significant for participants with motor impairments ($\chi^2_{(2,N=33)}$=21.15, $p < .001$).

	Motor-impaired			Able-bodied		
	Ability-based	Baseline	Preference-based	Ability-based	Baseline	Preference-based
Efficiency	1.48	2.61	1.91	1.71	2.29	2.00
Overall rank	1.64	2.48	1.88	1.95	2.00	2.05

Table 5: Average subjective ranking by efficiency and overall preference (1=best, 3=worst)

With respect to overall preference, participants with motor impairments significantly preferred the two personalized types of interfaces than the baselines ($\chi^2_{(2,N=33)}$=12.61, $p < .01$). Able-bodied participants had no detectable preference for any of the interface variants.

9. Discussion

Despite more than two decades of research on model-based automatic user interface generation, there remains a lot of skepticism about the very idea of automatic interface generation. In this section, we explicitly address some of the common concerns and indicate the novel aspects of our work that make it likely to have practical impact.

Automatically generated user interfaces are not as good as those created by human designers. What is the value of systems like SUPPLE?

Automatically generated user interfaces are typically perceived as being less aesthetically pleasing than those created by human designers [56]. Indeed, we do believe that hand-crafted user interfaces, which reflect designers' creativity and understanding of applications' semantics, will—for typical users in typical situations—result in more desirable interfaces than those created by automated tools. SUPPLE, therefore, is not intended to replace or compete with human designers. Instead, SUPPLE offers alternative user interfaces for those users whose devices, tasks, preferences, and abilities are not sufficiently addressed by the mainstream hand-crafted designs. Because there exist a myriad of distinct individuals, each with his or her own devices, tasks, preferences, and abilities, the problem of providing each person with the most appropriate interface is simply one of scale: there are not enough human experts to provide each user with an interface reflecting that person's context. The results of our user study demonstrate that people with motor impairments both perform better with and strongly prefer interfaces generated by SUPPLE compared to the manually designed default interfaces.

Our approach stands in contrast to the majority of prior work on model-based user interface design, where the automated design tools were used primarily as a means to incrementally improve existing design processes.

The creation of model-based user interfaces requires a large amount of upfront effort. This model creation is incompatible with the current design practice.

Indeed, nearly all model-based user interface toolkits require that users begin the UI design process by creating abstract models of the tasks or data (or both). Even if a system provides a graphical environment for designing such models (as does TERESA [67], for example), this is still inconsistent with the current design practice, which stresses the importance of exploring the space of *concrete* (even if low fidelity) designs from the very beginning of the design process [10, 48, 74, 78]. This high up-front cost has been identified as an important barrier to adoption of automatic user interface generation technology [56], and it turns user interface design into an abstract programming-like task, which is not our intention.

Instead, we believe that interfaces for typical users in typical situations should continue to be created by expert designers using current design methods. The abstract interface model should be automatically *inferred* as the designer creates and explores the concrete designs for the typical user. Indeed, this approach has been attempted in a recent system called Gummy [54]. Gummy observes the designer as he or she creates the default version of a user interface and it then automatically suggests designs for alternative platforms. We intend to develop such a design too, which—through only a small amount of additional interaction with the designer—will capture his or her rationale and design preferences, so that they can be reflected in the automatically generated alternatives.

Alternatively, the specification can be obtained by automatically reverse engineering a concrete user interface. The feasibility of this approach has been previously demonstrated for traditional (non-AJAX) web sites [9, 64] and more recently for desktop user interfaces [12, 45]. While some manual intervention will be required to refine such automatically extracted specifications, this approach may significantly reduce the barrier to automatically generating alternative user interfaces for existing applications.

Are systems like SUPPLE practical?

The most important limitation to the practical deployment of systems like SUPPLE is the current software engineering practice, which makes the user interface code inseparable from the application logic. HTML-based web applications are still an exception. It is therefore our intention to deploy our technology first in the web context, most likely as a JavaScript library that can be included with existing pages and applications to enable a rich set of client-side adaptations and customizations.

Is SUPPLE's approach limited to dialog box–like interfaces?

Our approach of casting user interface design as a discrete combinatorial problem is particularly well suited for dialog box–like user interfaces because there is a well-established vocabulary of interactions used for designing such interfaces. The approach is not limited to such interfaces, however. We hypothesize that it may also be possible to identify an analogous discrete set of basic operations for most canvas-based interfaces, such as word processors or image manipulation programs. We are encouraged by the results of a recent project that identified a vocabulary of 27 operations forming the foundation for most interactions performed on multi-touch surfaces [91].

What about documentation and tech support?

If systems like SUPPLE were to be widely adopted, what would happen to our ability to share expertise via documentation or other technical support mechanisms? For documentation, the answer is easy: it is trivial to automatically generate instructions showing the sequence of UI operations and to illustrate these instructions with automatically created screen shots. For remote technical support, where screen sharing is currently used, a "model sharing" approach could be used instead: the user's and the technician's versions of the software could be linked not at the level of the pixels, but at the level of the underlying model: the technician and the user can see different surface presentation of the application, but both would be operating identical functionality. If the technician, for example, set a combo box to a particular value, the same operation could be visualized on a user's screen regardless of how this functionality is rendered.

10. Conclusion

We have presented Supple, a system that automatically generates graphical user interfaces given a functional user interface specification, a model of the capabilities and limitations of the device, a cost function, and an optional usage model reflecting how the interface will be used. Supple naturally generates user interfaces adapted to different devices as well as varied motor abilities. It also provides mechanisms for automatic system-driven adaptation to both long-term and short-term usage patterns. As a complement to automatic generation and adaptation, Supple also supports an extensive user-driven customization mechanism that lets users modify the overall structure and individual pieces of any Supple-generated user interface. We illustrated our approach with a concrete application of Supple: automatically generating user interfaces adapted to the individual abilities of users with motor impairments.

Supple's optimization algorithm can generate user interfaces in less than a second in most cases, provided the cost function is expressed in a particular parametrized form. We have also introduced an alternative cost function formulation that can reflect user's motor capabilities, but which results in slower system performance (on the order of tens of minutes). An important consequence of casting user interface generation as an optimization problem is that the style of the user interfaces generated by Supple can be entirely determined by the appropriate parameterization of the cost functions. This offers the potential for personalizing the interface generation process. Consequently, we have subsequently developed two additional systems: Arnauld for eliciting users' subjective preferences [22] and Ability Modeler for modeling objective motor abilities [27].

The results of the summative user study, which involved 11 participants with motor impairments and 6 able-bodied participants, showed that the participants were significantly faster and made far fewer errors using the automatically generated personalized interfaces than with the default user interfaces. Additionally, participants with motor impairments strongly preferred automatically generated user interfaces to the default ones. By helping improve their efficiency, Supple helped narrow the gap between motor-impaired and able-bodied users by 62%, with individual gains ranging from 32% to 103%. These results demonstrate that the technical contributions presented in this paper have a potential to make a significant impact in practice.

In our work, we considered two metrics for optimizing user interfaces, namely, a model of users' preferences, and a model of their motor abilities. Future work should explore other individual metrics, such as those related to cognition and attention. But another interesting direction would be to consider metrics that reflect how different interface designs encourage or facilitate particular user behaviors. For example, an on-line merchant may wish for an interface that maximizes the number of product pages that a visitor explores, while a collaborative knowledge sharing site will benefit from maximizing the number and quality of knowledge contributions. Kohavi et al., [43] offer some helpful initial insights.

Another promising direction will be to pursue the semantic adaptation of user interfaces. In contrast to our work so far, where we adapted the *structure and presentation* of the interfaces, future work could explore ways to automatically adapt the *functionality* itself; that is, ways to automatically simplify user interfaces. This is an important problem because solving it would enable complex applications to be transformed for easier use on mobile devices and by users with cognitive impairments. It also would allow automatic generation of interfaces for novice users, and allow frequent users to quickly create task-specific simplified views of a complex interface. Such simplified interface views have been shown to significantly improve users' satisfaction, but are time-consuming to create and maintain by hand [52]. This is a hard problem to solve automatically, because it requires an understanding of the function and purpose of interface elements. The existing solutions rely on extensive semantic annotations by the designer or by the user [16]. An alternative approach would be to leverage large user communities by automatically mining usage and customization traces.

Supple is not intended to replace human designers. Instead, it can provide alternative user interfaces for those users whose individual circumstances are not sufficiently addressed by the hand-crafted designs. Because there exist a myriad of distinct individuals, each with his or her own devices, tasks, preferences, and abilities, the problem of providing each person with the most appropriate interface is simply one of scale: there are not enough human experts to provide each user with an interface reflecting that person's context. Our work demonstrates that automated tools are a feasible way of addressing this scalability challenge: the Supple system can generate user interfaces in a matter of seconds, and all the personalization mechanisms we subsequently developed rely entirely on user input, not requiring any expert assistance.

11. Acknowledgments

Kiera Henning contributed to the implementation of SUPPLE's HTML widget library and Jing Jing Long contributed to the AWT library used on the PDAs [19]. Jing Jing Long also initiated the implementation of the extensions to the Java Metal GUI toolkit [26] that allowed the target size and the visual cue size to be parametrized. Anthony Wu contributed to the design and implementation of SUPPLE's consistency mechanisms [23], and the initial implementation of SUPPLE's customization facilities was done by Raphael Hoffmann [20, 19].

The authors extend their thanks for advice, inspiration, and helpful feedback to Eytan Adar, Mark Adler, Alan Borning, Gaetano Borriello, Anna Cavender, Dan Comden, Rebecca Cremona, Mark Harniss, Curt Johnson, Kurt L. Johnson, James Landay, Tessa Lau, Jeffrey Nichols, Joe Stuckey, Steve Wolfman, and Alexander Yates.

This work was funded in part by a Microsoft Graduate Research Fellowship, Intel Research, NSF grant IIS-0307906, ONR grant N00014-06-1-0147, DARPA project CALO through SRI grant number 03-000225, and the WRF / TJ Cable Professorship. Parts of this manuscript were prepared when the first author was a postdoctoral researcher at Microsoft Research.

References

[1] Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, and Jonathan E. Shuster. UIML: an appliance-independent XML user interface language. In *WWW '99: Proceeding of the eighth international conference on World Wide Web*, pages 1695–1708, New York, NY, USA, 1999. Elsevier North-Holland, Inc.

[2] Johnny Accot and Shumin Zhai. Refining Fitts' law models for bivariate pointing. In *CHI '03: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 193–200, New York, NY, USA, 2003. ACM Press.

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

[4] J. Aitchison. *The Lognormal distribution*. Cambridge: Cambridge University Press, 1976.

[5] E. Bergman and E. Johnson. Towards Accessible Human-Computer Interaction. *Advances in Human-Computer Interaction*, 5(1), 1995.

[6] F. Bodart, A. M. Hennebert, J. M. Leheureux, I. Provot, B. Sacre, and J. Vanderdonckt. Towards a Systematic Building of Software Architectures: the TRIDENT Methodological Guide. *Design, Specification and Verification of Interactive Systems. Wien: Springer*, pages 262–278, 1995.

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

[8] Alan Borning, Kim Marriott, Peter Stuckey, and Yi Xiao. Solving linear arithmetic constraints for user interface applications. In *Proceedings of the 1997 ACM Symposium on User Interface Software and Technology*, pages 87–96, October 1997.

[9] Laurent Bouillon, Jean Vanderdonckt, and Kwok Chieu Chow. Flexible re-engineering of web sites. In *IUI '04: Proceedings of the 9th international conference on Intelligent user interfaces*, pages 132–139, New York, NY, USA, 2004. ACM.

[10] Bill Buxton. *Sketching User Experiences: Getting the Design Right and the Right Design (Interactive Technologies)*. Morgan Kaufmann, April 2007.

[11] M. Dawe. Desperately seeking simplicity: how young adults with cognitive disabilities and their families adopt assistive technologies. *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 1143–1152, 2006.

[12] Morgan Dixon and James Fogarty. Prefab: Implementing Advanced Behabiors Using Pixel-Based Reverse Engineering of Interface Structure. In *Proceedings of CHI '10*, New York, NY, USA, 2010. ACM.

[13] CS Fichten, M. Barile, JV Asuncion, and ME Fossey. What government, agencies, and organizations can do to improve access to computers for postsecondary students with disabilities: recommendations based on Canadian empirical data. *Int J Rehabil Res*, 23(3):191–9, 2000.

[14] Leah Findlater and Joanna McGrenere. A comparison of static, adaptive, and adaptable menus. In *Proceedings of ACM CHI 2004*, pages 89–96, 2004.

[15] P. M. Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *J Exp Psychol*, 47(6):381–391, June 1954.

[16] Murielle Florins, Francisco Montero Simarro, Jean Vanderdonckt, and Benjamin Michotte. Splitting rules for graceful degradation of user interfaces. In *AVI '06: Proceedings of the working conference on Advanced visual interfaces*, pages 59–66, New York, NY, USA, 2006. ACM.

[17] James Fogarty, Jodi Forlizzi, and Scott E. Hudson. Aesthetic information collages: generating decorative displays that contain information. In *UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 141–150, New York, NY, USA, 2001. ACM.

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

[19] Krzysztof Gajos, David Christianson, Raphael Hoffmann, Tal Shaked, Kiera Henning, Jing Jing Long, and Daniel S. Weld. Fast and robust interface generation for ubiquitous applications. In *Proceedings of Ubicomp'05*, Tokyo, Japan, 2005.

[20] Krzysztof Gajos, Raphael Hoffmann, and Daniel S. Weld. Improving user interface personalization. In *Supplementary Proceedings of UIST'04*, Santa Fe, NM, 2004.

[21] Krzysztof Gajos and Daniel S. Weld. Supple: automatically generating user interfaces. In *Proceedings of the 9th international conference on Intelligent user interface*, pages 93–100, Funchal, Madeira, Portugal, 2004. ACM Press.

[22] Krzysztof Gajos and Daniel S. Weld. Preference elicitation for interface optimization. In *Proceedings of UIST 2005*, Seattle, WA, USA, 2005.

[23] Krzysztof Gajos, Anthony Wu, and Daniel S. Weld. Cross-device consistency in automatically generated user interfaces. In *Proceedings of Workshop on Multi-User and Ubiquitous User Interfaces (MU3I'05)*, 2005.

[24] Krzysztof Z. Gajos, Mary Czerwinski, Desney S. Tan, and Daniel S. Weld. Exploring the design space for adaptive graphical user interfaces. In *AVI '06: Proceedings of the working conference on Advanced visual interfaces*, pages 201–208, New York, NY, USA, 2006. ACM Press.

[25] Krzysztof Z. Gajos, Katherine Everitt, Desney S. Tan, Mary Czerwinski, and Daniel S. Weld. Predictability and accuracy in adaptive user interfaces. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 1271–1274, New York, NY, USA, 2008. ACM.

[26] Krzysztof Z. Gajos, Jing Jing Long, and Daniel S. Weld. Automatically generating custom user interfaces for users with physical disabilities. In *Assets '06: Proceedings of the 8th international ACM SIGACCESS conference on Computers and accessibility*, pages 243–244, New York, NY, USA, 2006. ACM.

[27] Krzysztof Z. Gajos, Jacob O. Wobbrock, and Daniel S. Weld. Automatically generating user interfaces adapted to users' motor and vision capabilities. In *UIST '07: Proceedings of the 20th annual ACM symposium on User interface software and technology*, pages 231–240, New York, NY, USA, 2007. ACM Press.

[28] Krzysztof Z. Gajos, Jacob O. Wobbrock, and Daniel S. Weld. Improving the performance of motor-impaired users with automatically-generated, ability-based interfaces. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 1257–1266, New York, NY, USA, 2008. ACM.

[29] Saul Greenberg. *The computer user as toolsmith: The use, reuse, and organization of computer-based tools*. Cambridge University Press, 1993.

[30] T. Grossman and R. Balakrishnan. A probabilistic approach to modeling two-dimensional pointing. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 12(3):435–459, 2005.

[31] Susumu Harada, Jacob O. Wobbrock, and James A. Landay. Voicedraw: A voice-driven hands-free drawing application. In *ASSETS'07*. ACM Press, 2007.

[32] A.K. Hartmann and M. Weigt. *Phase Transitions in Combinatorial Optimization Problems: Basics, Algorithms and Statistical Mechanics*. Wiley-VCH, 2006.

[33] Alexander K. Hartmann and Martin Weigt. Statistical mechanics perspective on the phase transition in vertex covering of finite-connectivity random graphs. *Theoretical Computer Science*, 265(1-2):199–225, August 2001.

[34] Philip J. Hayes, Pedro A. Szekely, and Richard A. Lerner. Design alternatives for user interface management sytems based on experience with cousin. In *CHI '85: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 169–175, New York, NY, USA, 1985. ACM.

[35] Ken Hinckley, Edward Cutrell, Steve Bathiche, and Tim Muss. Quantitative analysis of scrolling techniques. In *CHI '02: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 65–72, New York, NY, USA, 2002. ACM Press.

[36] Todd D. Hodes, Randy H. Katz, Edouard Servan-Schreiber, and Lawrence Rowe. Composable ad-hoc mobile services for universal interaction. In *MobiCom '97: Proceedings of the 3rd annual ACM/IEEE international conference on Mobile computing and networking*, pages 1–12, New York, NY, USA, 1997. ACM.

[37] S. Holm. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 6(65-70):1979, 1979.

[38] A. Hornof, A. Cavender, and R. Hoselton. Eyedraw: a system for drawing pictures with eye movements. *Proceedings of the ACM SIGACCESS conference on Computers and accessibility*, pages 86–93, 2004.

[39] Faustina Hwang, Simeon Keates, Patrick Langdon, and John Clarkson. Mouse movements of motion-impaired users: a submovement analysis. In *Assets '04: Proceedings of the 6th international ACM SIGACCESS conference on Computers and accessibility*, pages 102–109, New York, NY, USA, 2004. ACM Press.

[40] Christian Janssen, Anette Weisbecker, and Jürgen Ziegler. Generating user interfaces from data models and dialogue net specifications. In *CHI '93: Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, pages 418–423, New York, NY, USA, 1993. ACM.

[41] Simeon Keates, Patrick Langdon, John P. Clarkson, and Peter Robinson. User models and user physical capability. *User Modeling and User-Adapted Interaction*, 12(2):139–169, June 2002.

[42] H.H. Koester. Abandonment of speech recognition by new users. *Proceedings of the 26th Annual Conference of the Rehabilitation Engineering and Assistive Technology Society of North America (RESNA03). Atlanta, Georgia (June 19-23, 2003)*, 2003.

[43] Ron Kohavi, Randal M. Henne, and Dan Sommerfield. Practical guide to controlled experiments on the web: listen to your customers not to the HiPPO. In *KDD '07: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 959–967, New York, NY, USA, 2007. ACM.

[44] B. Lafreniere, A. Bunt, J. Whissell, C. Clarke, and M. Terry. Characterizing large-scale use of a direct manipulation application in the wild. Technical Report CS-2009-27, David R. Cheriton School of Computer Science, University of Waterloo, 2009.

[45] F Lamberti and A Sanna. Extensible GUIs for Remote Application Control on Mobile Devices. *Computer Graphics and Applications, IEEE*, 28(4):50–57, 2008.

[46] CM Law, A. Sears, and KJ Price. Issues in the categorization of disabilities for user testing. In *Proceedings of HCII*, 2005.

[47] E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.

[48] James Lin and James A. Landay. Employing patterns and layers for early-stage design and prototyping of cross-device user interfaces. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 1313–1322, New York, NY, USA, 2008. ACM.

[49] R.C. Littel, G.A. MIlliken, W.W. Stroup, and R.D. Wolfinger. *SAS System for Mixed Models*. SAS Institute, Inc., Cary, NC, 1996.

[50] Wendy E. Mackay. Triggers and barriers to customizing software. In *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 153–160, New York, NY, USA, 1991. ACM Press.

[51] Scott I. MacKenzie and William Buxton. Extending fitts' law to two-dimensional tasks. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 219–226, New York, NY, USA, 1992. ACM Press.

[52] Joanna McGrenere, Ronald M. Baecker, and Kellogg S. Booth. An evaluation of a multiple interface design solution for bloated software. In *CHI '02: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 164–170, New York, NY, USA, 2002. ACM Press.

[53] Joanna McGrenere, Ronald M. Baecker, and Kellogg S. Booth. A field evaluation of an adaptable two-interface design for feature-rich software. *ACM Trans. Comput.-Hum. Interact.*, 14(1), May 2007.

[54] Jan Meskens, Jo Vermeulen, Kris Luyten, and Karin Coninx. Gummy for multi-platform user interface designs: Shape me, multiply me, fix me, use me. In *AVI'08*. ACM Press, 2008.

[55] L. G. Mitten. Branch-and-bound methods: General formulation and properties. *Operations Research*, 18(1):24–34, 1970.

[56] Brad Myers, Scott E. Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7(1):3–28, March 2000.

[57] Jeffrey Nichols, Brad A. Myers, Michael Higgins, Joe Hughes, Thomas K. Harris, Roni Rosenfeld, and Mathilde Pignol. Generating remote control interfaces for complex appliances. In *CHI Letters: ACM Symposium on User Interface Software and Technology, UIST'02*, Paris, France, 2002.

[58] Jeffrey Nichols, Brad A. Myers, and Brandon Rothrock. UNIFORM: automatically generating consistent remote control user interfaces. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 611–620, New York, NY, USA, 2006. ACM Press.

[59] Jeffrey Nichols, Brandon Rothrock, Duen H. Chau, and Brad A. Myers. HUDDLE: automatically generating interfaces for systems of multiple connected appliances. In *UIST '06: Proceedings of the 19th annual ACM symposium on User interface software and technology*, pages 279–288, New York, NY, USA, 2006. ACM Press.

[60] Stina Nylander, Markus Bylund, and Annika Waern. The ubiquitous interactor - device independent access to mobile services. In *proceedings of the conference on Computer Aided Design of User Interfaces (CADUI'2004)*, Funchal, Portugal, 2004.

[61] D. R. Olsen. A programming language basis for user interface. In *CHI '89: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 171–176, New York, NY, USA, 1989. ACM Press.

[62] Dan R. Olsen, Sean Jefferies, Travis Nielsen, William Moyes, and Paul Fredrickson. Cross-modal interaction using xweb. In *UIST '00: Proceedings of the 13th annual ACM symposium on User interface software and technology*, pages 191–200, New York, NY, USA, 2000. ACM.

[63] Reinhard Oppermann and Helmut Simm. Adaptability: user-initiated individualization. *Adaptive user support: ergonomic design of manually and automatically adaptable software table of contents*, pages 14–66, 1994.

[64] L. Paganelli and F. Paterno. Automatic reconstruction of the underlying interaction design of web applications. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 439–445, New York, NY, USA, 2002. ACM.

[65] Leysia Palen. Social, individual and technological issues for groupware calendar systems. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 17–24, New York, NY, USA, 1999. ACM.

[66] Fabio Paterno, Cristiano Mancini, and Silvia Meniconi. ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. In *INTERACT*, pages 362–369, 1997.

[67] Fabio Paterno, Carmen Santoro, Jani Mantyjarvi, Giulio Mori, and Sandro Sansone. Authoring pervasive multimodal user interfaces. *Int. J. Web Eng. Technol.*, 4(2):235–261, 2008.

[68] B. Phillips and H. Zhao. Predictors of assistive technology abandonment. *Assist Technol*, 5(1):36–45, 1993.

[69] Shankar Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd. ICrafter: A service framework for ubiquitous computing environments. In *Proceedings of Ubicomp 2001*, pages 56–75, 2001.

[70] Patrick Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81(1-2):81–109, March 1996.

[71] David Reitter, Erin Panttaja, and Fred Cummins. UI on the fly: Generating a multimodal user interface. In *Proceedings of Human Language Technology conference 2004 / North American chapter of the Association for Computational Linguistics (HLT/NAACL-04)*, 2004.

[72] Kai Richter, Jeffrey Nichols, Krzysztof Gajos, and Ahmed Seffah, editors. *Proceedings of the CHI'06 Workshop on The Many Faces of Consistency in Cross Platform Design*, 2006.

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

[74] Dan Saffer. *Designing for Interaction: Creating Smart Applications and Clever Devices (Voices That Matter)*. Peachpit Press, 1 edition, July 2006.

[75] Christof Schuster and Alexander Von Eye. The Relationship of ANOVA Models with Random Effects and Repeated Measurement Designs. *Journal of Adolescent Research*, 16(2):205–220, March 2001.

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

[77] Andrew Sears and Ben Shneiderman. Split menus: effectively using selection frequency to organize menus. *ACM Trans. Comput.-Hum. Interact.*, 1(1):27–51, 1994.

[78] Helen Sharp, Yvonne Rogers, and Jenny Preece. *Interaction Design: Beyond Human-Computer Interaction*. Wiley, 2 edition, March 2007.

[79] Piyawadee Sukaviriya, James D. Foley, and Todd Griffith. A second generation user interface design environment: the model and the runtime architecture. In *CHI '93: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 375–382, New York, NY, USA, 1993. ACM Press.

[80] Pedro Szekely, Ping Luo, and Robert Neches. Facilitating the exploration of interface design alternatives: the humanoid model of interface design. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 507–515, New York, NY, USA, 1992. ACM.

[81] Pedro A. Szekely, Piyawadee N. Sukaviriya, Pablo Castells, Jeyakumar Muthukumarasamy, and Ewald Salcher. Declarative interface models for user interface construction tools: the mastermind approach. In *Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction*, pages 120–150, London, UK, UK, 1996. Chapman & Hall, Ltd.

[82] J. Vanderdonckt and F. Bodart. The “Corpus Ergonomicus”: A Comprehensive and Unique Source for Human-Machine Interface Guidelines, in” Advances in Applied Ergonomics. In *Proceedings of 1st International Conference on Applied Ergonomics ICAE*, pages 162–169, 1996.

[83] Jean M. Vanderdonckt and François Bodart. Encapsulating knowledge for intelligent automatic interaction objects selection. In *CHI ’93: Proceedings of the INTERACT ’93 and CHI ’93 conference on Human factors in computing systems*, pages 424–429, New York, NY, USA, 1993. ACM.

[84] J. K. Vermunt. *Log-linear Models for Event Histories*. Sage Publications, 1997.

[85] Ian Vollick, Daniel Vogel, Maneesh Agrawala, and Aaron Hertzmann. Specifying label layout style by example. In *UIST ’07: Proceedings of the 20th annual ACM symposium on User interface software and technology*, pages 221–230, New York, NY, USA, 2007. ACM.

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

[87] Charles Wiecha, William Bennett, Stephen Boies, John Gould, and Sharon Greene. ITS: a tool for rapidly developing interactive applications. *ACM Transactions on Information Systems (TOIS)*, 8(3):204–236, 1990.

[88] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[89] Colin P. Williams and Tad Hogg. Exploiting the deep structure of constraint problems. *Artif. Intell.*, 70(1-2):73–117, 1994.

[90] Christopher Winship and Robert D. Mare. Regression models with ordinal variables. *American Sociological Review*, 49(4):512–525, 1984.

[91] Jacob O. Wobbrock, Meredith Ringel Morris, and Andrew D. Wilson. User-defined gestures for surface computing. In *CHI ’09: Proceedings of the 27th international conference on Human factors in computing systems*, pages 1083–1092, New York, NY, USA, 2009. ACM.

[92] Brad V. Zanden and Brad A. Myers. Automatic, look-and-feel independent dialog creation for graphical user interfaces. In *CHI ’90: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 27–34, New York, NY, USA, 1990. ACM.

[93] Michelle X. Zhou and Vikram Aggarwal. An optimization-based approach to dynamic data content selection in intelligent multimedia interfaces. In *UIST ’04: Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 227–236. ACM Press, 2004.

[94] Michelle X. Zhou, Zhen Wen, and Vikram Aggarwal. A graph-matching approach to dynamic media allocation in intelligent multimedia interfaces. In *IUI ’05: Proceedings of the 10th international conference on Intelligent user interfaces*, pages 114–121. ACM Press, 2005.