Automatically Generating Personalized User Interfaces

Krzysztof Z. Gajos

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

2008

Program Authorized to Offer Degree: Computer Science and Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Krzysztof Z. Gajos

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of the Supervisory Committee:

_____

Daniel S. Weld

Reading Committee:

_____

Daniel S. Weld

_____

James A. Landay

_____

Jacob O. Wobbrock

Date: _____

In presenting this dissertation in partial fulfillment of the requirements for the doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Proquest Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, 1-800-521-0600, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature_____

Date_____

University of Washington

Abstract

Automatically Generating Personalized User Interfaces

Krzysztof Z. Gajos

Chair of the Supervisory Committee:
Professor Daniel S. Weld
Computer Science and Engineering

User Interfaces for today's software are usually created in a one-size-fits-all manner, making implicit assumptions about the needs, abilities, and preferences of the "average user" and the characteristics of the "average device." I argue that personalized user interfaces, which are adapted to a person's devices, tasks, preferences, and abilities, can improve user satisfaction and performance. I have developed three systems:

- SUPPLE, which uses decision-theoretic optimization to automatically generate user interfaces adapted to a person's device and long-term usage;
- ARNAULD, which allows optimization-based systems to be adapted to users' preferences; and
- ABILITY MODELER and an extension of SUPPLE that first performs a one-time assessment of a person's motor abilities and then automatically generates user interfaces predicted to be the fastest to use for that user.

My experiments show that these automatically generated, personalized user interfaces significantly improve speed, accuracy, and satisfaction for users with motor impairments compared to manufacturers' default interfaces. I also provide the first characterization of the design space of adaptive graphical user interfaces, and demonstrate how such interfaces can significantly improve the quality and efficiency of daily interactions for typical users.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to begin by acknowledging my student collaborators. Raphael Hoffmann created the initial implementation of SUPPLE's customization infrastructure. Anthony Wu contributed to the development of the framework for cross-device consistency of automatically generated user interfaces. Kiera Henning and Jing Jing Long implemented most of the extensions that allowed SUPPLE to generate user interfaces for web browsers and PDAs, respectively. Jing Jing also helped explore the approaches for modifying the Java Swing Metal Look and Feel so that it could be used to support the needs of users with impaired dexterity and low vision. Finally, Katherine Everitt and I together designed and executed one of the user studies exploring the design space of adaptive user interfaces.

I thank Donald Patterson and Brian DeRenzi for the collaboration opportunities, and Anna Cavender for helping me realize that a user interface designed with a person's abilities in mind can enable fruitful and enjoyable interaction with computers.

It was a real pleasure to be a graduate student at the Computer Science and Engineering department at the University of Washington. Its faculty, students, and staff made it a supportive, exciting, and inspiring environment. In particular, Lindsay Michimoto's thoughtful guidance helped to make my graduate school experience smooth and enjoyable. I also thank Tessa Lau, who in many long conversations shared with me the secret wisdom of a successful graduate student.

The conversations with Batya Friedman and the Value Sensitive Design group broadened my understanding of how technology impacts our society.

The generous and patient mentorship of Mary Czerwinski and Desney Tan at Microsoft Research was crucial to my development as an HCI researcher. I am particularly grateful to Mary for having faith in my ideas and opening the doors of her lab to me. James Landay provided pointed and extensive feedback at several stages of the project. I have also

benefited from the helpful advice of Gaetano Borriello, Oren Etzioni and Pedro Domingos throughout my six years at the University of Washington. Kurt Johnson, Mike Harniss, and Curt Johnson at the UW Department of Rehabilitation Medicine offered critical assistance with designing the final user study and introducing me to potential participants.

Lastly, I want to thank my advisors Dan Weld and Jake Wobbrock. Although I only had the benefit of Jake's advice for the last two years of my graduate career, his expertise was essential for developing the Ability Modeler and conducting the final evaluation with users with motor impairments. The experimental design skills, rigorous data analysis techniques, and attention to detail that I learned from him will benefit me for the rest of my professional life. I have no doubt that I would not be where I am today if it were not for Dan Weld. He offered me mentorship and friendship that helped me grow as a researcher and as a person. His curiosity about my ideas gave me the confidence to pursue them. He taught me the critical skill of how to rigorously and formally structure my findings and convey them to others. He offered close guidance yet made me feel that I was in charge. He also introduced me to the fascinating world of canyoneering.

Finally, I want to thank my sister Kasia for her friendship, my parents for a home where my curiosity was never left unsatisfied, and Dr. Jean for happiness, sanity, and everything.

Chapter 1

INTRODUCTION

Today's user interfaces are typically designed with the assumption that they are going to be used by an able-bodied user, who has typical perceptual and cognitive abilities, who is sitting in a stable warm environment, and who is using a typical set of input and output devices. Any deviation from these assumptions (for example, hand tremor due to aging, low vision, riding on a jostling bus, trying to use a laser pointer to control a mouse cursor), may drastically hamper users' effectiveness — not because of any *inherent* barrier to interaction, but because of a mismatch between users' effective abilities and the assumptions underlying the user interface design.

Currently, this diversity of needs is either ignored or it is addressed in one of two ways: a manual redesign of a user interface (e.g., for use on a new platform), or with an external assistive technology (e.g., for users with motor or perceptual impairments) that provide a mechanism for users to adapt to user interfaces. The first approach is clearly not scalable: new devices constantly enter the market, and people's abilities and preferences differ greatly, and often cannot be anticipated in advance [10]. The second approach, which often enables computer access to people who would otherwise not have it, also has limitations: assistive technologies can stigmatize their users; they are also impractical for people with temporary impairments caused by injuries; they do not adapt to users whose abilities change over time; and finally, they are often abandoned, even by users who need them, because of factors like cost, complexity, configuration, and the need for ongoing maintenance [29, 31, 75, 113].

Other approaches that try to address the diversity of abilities, like universal design [84], inclusive design [72], and design for all [134], attempt to create technologies that have properties suitable to as many people as possible. This is a laudable goal, but these approaches are impractical in many cases, particularly where complex software systems are involved [10].

A "one size fits all" approach often cannot accommodate the broad range of abilities and skills in vast and varied user populations.

In contrast to these approaches, I argue that interfaces should be personalized to better suit the contexts of individual users. Many personalized interfaces are needed because of the myriad of distinct individuals, each with his or her own abilities, 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. It is therefore my thesis that:

automatically generated user interfaces, which are adapted to a person's devices, tasks, preferences, and abilities, can improve people's satisfaction and performance compared to traditional manually designed "one size fits all" interfaces.

In my dissertation work, I develop three systems to enable a broad range of personalized adaptive interfaces: SUPPLE, which uses decision-theoretic optimization to automatically generate user interfaces adapted to a person's device and usage; ARNAULD, which allows optimization-based systems to be adapted to users' preferences; and the ABILITY MODELER, which performs a one-time assessment of a person's motor abilities and then automatically builds a model of those abilities, which SUPPLE uses to automatically generate user interfaces adapted to that user's abilities. The results of my laboratory experiments show that these **automatically generated, ability-based user interfaces significantly improve speed, accuracy and satisfaction of users with motor impairments compared to manufacturers' defaults.** I also provide the first characterization of the design space of adaptive graphical user interfaces, and demonstrate how such interfaces can significantly improve the quality and efficiency of daily interactions for typical users.

The following sections provide a summary of the main contributions of this dissertation.

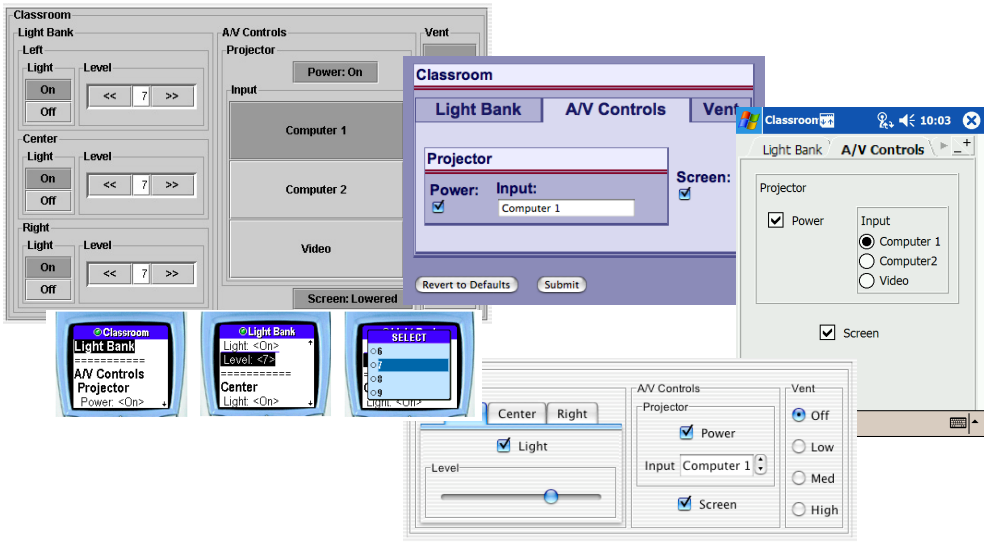


Figure 1.1: An interface for a simple application rendered automatically by SUPPLE for five different platforms: a touch panel, an HTML browser, a PDA, a desktop computer and a WAP phone.

1.1 Supple*: A Platform For Automatically Generating Personalized User Interfaces*

My SUPPLE system [41, 39], introduced in Chapter 3, uses constrained decision-theoretic optimization to automatically generate user interfaces. As input, SUPPLE takes a *functional specification* of the interface, which describes the types of information that need to be communicated between the application and the user, the device-specific constraints, such as screen size and a list of available interactors, a typical *usage trace*, and a *cost function*. The cost function can correspond to any measure of interface quality, such as expected ease of use or conformance to user's preferences. SUPPLE's optimization algorithm finds the user interface that minimizes the cost function while also satisfying all device constraints. Despite the huge space of possible designs (exponential in the number of interface elements), in the vast majority of cases **my algorithm renders even complex interfaces in less than a second** on a standard desktop computer, thus enabling fast and personalized delivery of user interfaces.

Although there is some previous work that used optimization methods for laying out widgets within a dialog box (e.g., [38, 128]), my rendering algorithm does much more: it chooses the widgets, it chooses some of the navigation structure of the UI (i.e., it separates content into tab panes or pop-up windows if not everything can fit on one screen), and it chooses the layout.

Unlike the previous approaches, which use rule-based techniques for UI generation (e.g., [106, 147, 71, 105]), SUPPLE naturally adapts to devices with vastly different screen sizes [41, 39]. Using SUPPLE on a novel type of a device requires only specifying a cost function and a new device model listing what widgets are available (Figure 1.1). Further, by modifying the cost function, SUPPLE can be made to produce very different styles of user interfaces or to accommodate other objectives, such as presentation consistency with previously generated versions of the interface, even if they were generated for a different device [43].

1.2 Arnauld*: Adapting To Users' Preferences*

By providing a different cost function, one can direct SUPPLE to produce very different styles of user interfaces. For example, by encoding a user's preferences in the cost function, SUPPLE will generate user interfaces that the user favors. However, SUPPLE's cost functions typically rely on more than 40 parameters reflecting complex decision trade-offs. Manual selection of the values of these parameters is a tedious and error-prone process. To address this challenge I built ARNAULD, a system described in Chapter 4 that quickly calculates parameter values from user feedback about concrete outcomes [42]. To do this, ARNAULD uses two types of interactions: *system-driven elicitation*, where the user is presented with pairwise comparison queries (Figure 1.2) and *user-driven example critiquing*, which relies on user-initiated improvements to the SUPPLE-generated interfaces as input to the learning system. These interactions allow people to express their preferences more easily and consistently than other common approaches such as ranking or rating [23] but they also necessitated development of a novel machine learning algorithm.

ARNAULD's learning algorithm uses a max-margin approach to find a set of parameters that optimally matches the preferences expressed by the user. Similar problems have been

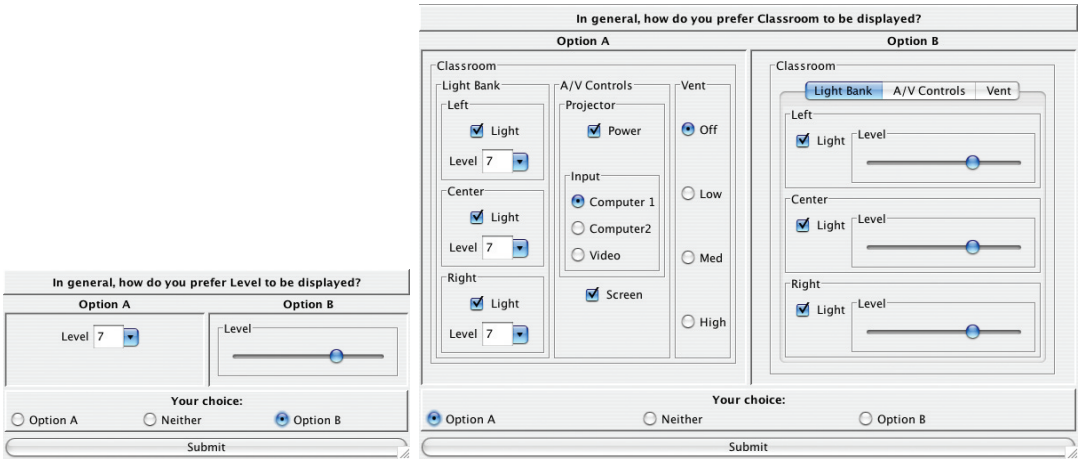


Figure 1.2: The first step in the active elicitation process: (left) ARNAULD first poses a *ceteris paribus* query, showing two renderings of light intensity control in isolation; this user prefers to use a slider. (right) Realizing that the choice may impact other parts of the classroom controller interface, ARNAULD subsequently asks the user to consider a concrete interface that uses combo boxes for light intensities but is able to show all elements at once, and an interface where sliders are used but different parts of the interface have to be put in separate tab panes in order to meet the overall size constraints.

addressed using Support Vector Machines [50], but this requires expensive quadratic optimization. Bayesian reasoning using Manhattan sampling has also been tried [24]. Because none of these approaches were fast enough for interactive use *I developed a faster algorithm, recasting the problem as linear optimization.*

The system-driven elicitation queries require ARNAULD to generate the most informative comparisons for the user's consideration — i.e., it is an active learning problem. I developed two heuristic query-generation algorithms: one general, which performs computation entirely in the parameter space, and the other domain-specific, which is more robust to inconsistencies in user responses. The user and algorithmic evaluations show that only 40-55 user responses are needed to learn a cost function that closely approximates the desired behavior.

Decision-theoretic optimization is becoming a popular tool in the user interface community (see, for example, LineDrive [3], Kandinsky [36], GADGET [38], or RIA [152, 153]), and creating accurate cost (or utility) functions has become a bottleneck for both personalizing and extending those systems. My ARNAULD system is a general tool that can be incorporated into other projects.

1.3 Adapting To Users' Motor And Vision Abilities

Users with motor impairments often find it difficult or impossible to use today's common software applications. While many believe that the needs of these users are adequately addressed by specialized assistive technologies, these technologies, while often helpful, have two major shortcomings. First, they are often abandoned, because of their cost, complexity, limited availability and need for configuration and ongoing maintenance (it is estimated that about 60% of the users who indicated need for assistive technologies actually use them [31]). Second, assistive technologies are designed on the assumption that the user interface, which was designed for the "average user," is immutable, and thus users with motor impairments must adapt *themselves* to these interfaces through specialized technologies.

In Chapter 5, I present an alternative approach: my ABILITY MODELER builds a personalized model of a person's motor abilities, which SUPPLE then uses to automatically generate interfaces tailored to that individual's actual motor abilities (Figure 1.3).

Specifically, the ABILITY MODELER performs automatic feature selection to generate an individual regression model to represent a person's motor abilities after the user performs a one-time motor performance test. Given this predictive model of a person's abilities, SUPPLE uses the expected time it would take the user to complete a set of typical tasks with the underlying application as the objective function in the optimization process. Thus, given an application and a set of typical tasks performed with that application, SUPPLE generates an interface that is predicted to be the fastest to use for that person.

In order to accommodate this new cost function, I significantly extended SUPPLE's optimization algorithm. Despite addressing a computationally much harder problem, my new algorithm generates interfaces in a matter of seconds or minutes.

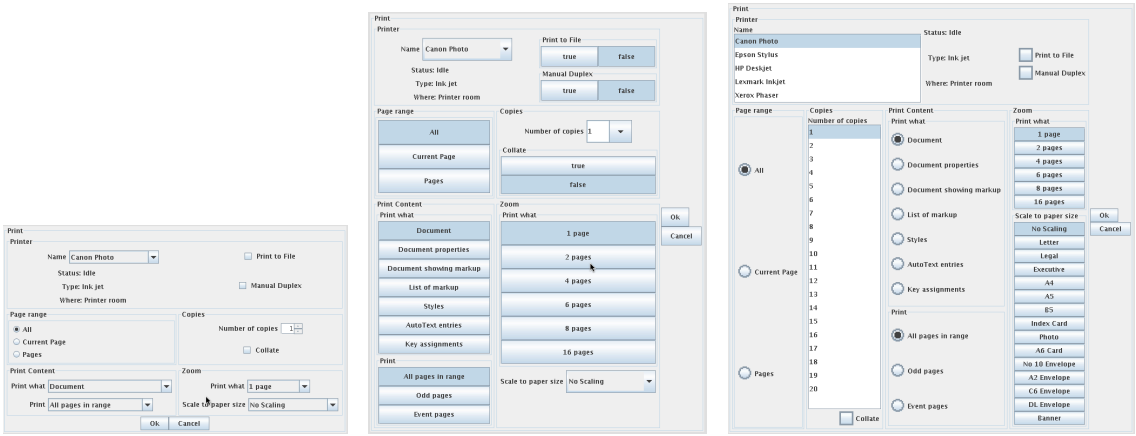


Figure 1.3: Three user interfaces for a print dialog box: a default reproduced from Microsoft Word 2003 and two automatically generated for a user with impaired dexterity: one based on his subjective preferences as elicited by ARNAULD, and one based on his actual motor abilities as modeled by the ABILITY MODELER.

In Chapter 5, I also describe a mechanism that allows SUPPLE users to adjust the interfaces to their vision capabilities. These two adaptations—to motor and vision capabilities—can also be used in combination. This is an important contribution because users with low vision *and* low dexterity are poorly supported by existing assistive technologies.

Besides producing adaptations for people with motor impairments, the approach of generating user interfaces optimized for a user's measured motor performance provides an approach for adapting interfaces to unusual devices or interaction techniques. For example, it provides a means to adapt user interfaces to touch panels or projected displays when using a laser pointer to control the mouse cursor.

1.3.1 Does It Matter?

This technology has the potential to provide great benefit. In a study involving 11 participants with motor impairments, which compared the interfaces generated by SUPPLE to manufacturer's default designs, results show that users with motor impairments were 26%

Figure 1.4: People with motor impairments who tried SUPPLE used a variety of strategies to control their pointing devices (the person in the top left corner used his chin).

faster using interfaces generated by SUPPLE, made 73% fewer errors, *strongly preferred* those interfaces to the manufacturers' defaults, and found them more efficient, easier to use, and much less physically tiring [49].

While some may contend that the needs of users with disabilities are adequately addressed by specialized assistive technologies, I have already argued these technologies are frequently abandoned—indeed, while we gave our participants the option to use any input devices they wanted, all of them chose either a mouse or a trackball (although, as illustrated in Figure 1.4, they used a variety of methods for using these devices). More fundamentally, assistive technologies are designed on the assumption that the user interface, which was designed for the "average user," is immutable, and thus users with motor impairments must adapt *themselves* to these interfaces by using specialized devices [73, 135].

In contrast, the results of my studies demonstrate that users with motor impairments can perform well with conventional input devices such as mice or trackballs when using interfaces that accommodate their unique motor abilities. These results also show that automatic generation of user interfaces based on users' motor abilities is feasible, and that the resulting interfaces are an attractive alternative to manufacturer's defaults.

1.4 The Design Space of Adaptive User Interfaces

Automatic adaptation of user interfaces is a contentious research area. Proponents (e.g., [9]) argue that it offers the potential to optimize interactions for a user's tasks and style, while critics (e.g., [32]) maintain that the inherent unpredictability of adaptive interfaces

may disorient the user, causing more harm than good. Surprisingly, however, little empirical evidence exists to inform this debate, and the existing research includes both positive and negative examples of adaptation, sometimes reporting contradictory results without analyzing the reasons underlying the discrepancy (e.g., [32] and [129]).

In Chapter 6, I describe three laboratory studies I have conducted with three very distinct techniques for adapting an interface to a person's current task at hand [44, 45]. By synthesizing my results with past research, **I provided the first characterization of the design space of the adaptive graphical user interfaces**, showing how different design and contextual factors affect users' performance and satisfaction. Specifically, I studied accuracy and predictability of the adaptive algorithm, the adaptation frequency, the frequency with which the user interacts with the interface, and the task complexity. My studies consistently showed that one existing approach to adaptation, which we termed Split Interfaces, results in a **significant improvement in both performance and satisfaction** compared to the non-adaptive baselines. In Split Interfaces, frequently-used functionality is copied to a specially designated adaptive part of the interface (see Figure 1.5, left).

1.5 Adapting to Individual Usage

Informed by my studies, I implemented the Split Interface approach in SUPPLE for adapting to the user's task at hand (Chapter 6). Unlike previous implementations of this general approach, which could only adapt contents of menu items or toolbar buttons, 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 [39]. For example, if the user frequently changes the print quality setting, which requires 4 to 6 mouse clicks to access in a typical print dialog box, SUPPLE will copy that functionality to the adaptive part of the main print dialog box (Figure 1.5, left).

As described already in Chapter 3, my decision-theoretic approach allows SUPPLE to also **adapt to a person's long term usage patterns**. By re-weighting the terms of the cost function in keeping with collected usage traces, SUPPLE generates interfaces where frequently used functionality gets rendered using more convenient widgets, and where ele-

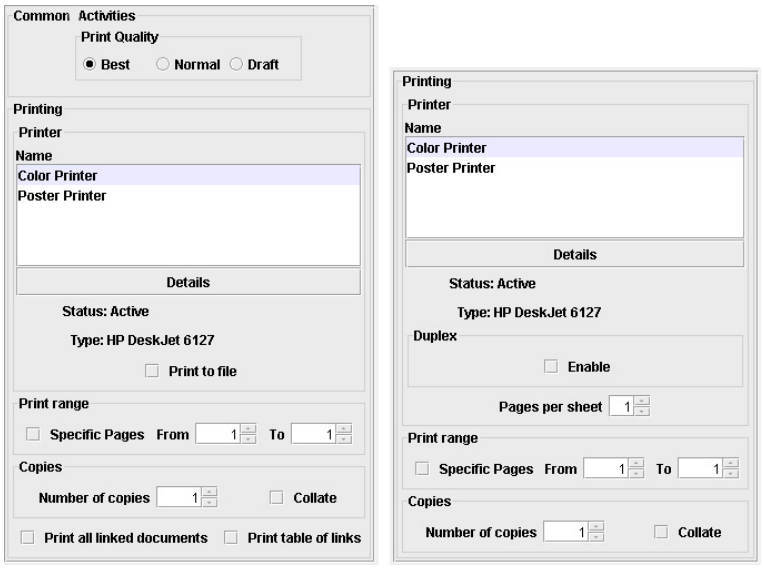


Figure 1.5: Two examples of personalization in SUPPLE: the left window features a dynamic section at the top whose automatically updated content reflects the most common activity; the right window was customized by the user, who removed some elements (e.g., "Print to file") and added "Duplex" and "Number of pages per sheet" elements by dragging them from a separate Printer Properties window.

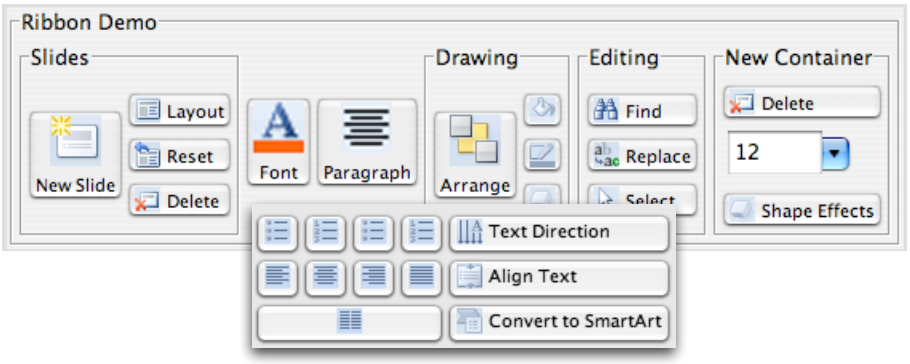


Figure 1.6: Unlike the manually designed Ribbon interface from Microsoft Office, the SUPPLE version allows users to add, delete, copy and move functionality; in this example, the New Container section was added, its contents copied via drag-and-drop operations from other parts of the interface, and the Quick Style button was removed from the Drawing panel; the customized SUPPLE version of the Ribbon can still adapt to different size constraints.

ments that are frequently used together are rendered side by side rather than in separate windows or panes [41].

Finally, SUPPLE supports extensive **user-driven customization**. With a right click of a mouse, users can change the presentation of a particular piece of functionality (e.g., request that a slider, rather than a pull-down menu, be used to represent sound volume), layout or navigation structure (e.g., request that particular set of items be rendered side-by-side rather than in separate tab panes or pop-up windows). These customizations are then represented as additional constraints during the rendering process. Users can also copy, move or delete parts of the interface. For example, instead of relying on automatic adaptation, a user can copy the duplex control for a printer from the "Properties" pop-up window to any place in the main print dialog (Figure 1.5, right). Those customizations are recorded as modifications to the functional specification graph and thus can be applied even when the interface is subsequently rendered on a different device.

This level of user control over the presentation of a user interface is not currently possible in manually designed user interfaces. A case in point is the Microsoft Ribbon, a new interface innovation introduced in Microsoft Office 2007. Ribbon, which replaces toolbars and menus, adapts to the width of the window but the adaptation is not automatic – several versions of the Ribbon were designed for different window sizes. An unfortunate consequence of this approach is that no manual customization of the Ribbon is possible. Unlike in toolbars from earlier versions of Microsoft Office, in Ribbon, there is no mechanism to allow moving, copying, adding or deleting buttons, panels or other functionality. My SUPPLE implementation of Ribbon [47] naturally supports both the adaptation to window width and a rich set of customizations, as illustrated in Figure 1.6.

Chapter 2

PREVIOUS WORK

The work presented in this dissertation builds upon prior research in a number of areas, including model-based user interface generation, preference elicitation, universal accessibility, and adaptive user interfaces. In this section, I review this prior work and indicate how it relates to my research.

2.1 Model-Based User Interface Generation

My 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 [59], Mickey [106], ITS [147], Jade [151], HUMANOID [137], UIDE [136], GENIUS [71], TRIDENT [143, 12], MASTERMIND [138], the "universal interaction" approach [62], XWeb [107], UIML [1], Personal Universal Controller [101] (and the related HUDDLE [103] and UNIFORM [102] projects), UI on the Fly [119], TERESA [111], TERESA [111], Ubiquitous Interactor [105]—dating back to the 1980's, that 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 [62], 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 or 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 users with impairments, but none of these projects resulted in any concrete solutions for such users. In summary, the 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 [114], 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 runtime generation of user interfaces, and of those only the Personal Universal Controller [101] (and the related HUDDLE and UNIFORM projects) resulted in a complete system while others (e.g., the "universal interaction" approach [62] or XWeb) assumed the existence of an external user interface generator.

Of those systems that provided some mechanism to automatically generate user interfaces, the majority used a very 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 [142] 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 system 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) [111]. 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 [110].

2.2 Optimization and Constraint Satisfaction For Interface Generation

Constraints have been used as a way to define flexible layouts that provided some level of device independence [13, 14]. 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 [36] 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 [152], and how to best match different pieces of information to different modalities [153]. Optimization is also a natural technique for automatically positioning labels in complex diagrams and visualizations [145]. Motivated by the growing use of optimization in automating parts of the interactive systems, the GADGET toolkit [38] 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 in the context of user interface generation only twice: by the GADGET toolkit and with the Layout Appropriateness user interface quality metric [128]. 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. In some cases the same optimization procedure is also used to find the optimal size for all the clickable elements in the interface.

2.3 Adapting Interfaces to User Preferences

Previous model-based interface generation systems generally did not provide any mechanisms for adapting the user interface generation process to the preferences of their end users. Although the authors of Jade [151] and MASTERMIND [138] systems recognized that this would be a desirable capability, they left it as future work. The TRIDENT system [143, 12], however, provided an interface for the designers to manually edit the interface generation rules. Finally, within the MOBI-D framework [117], a system was developed [30], which used the user's critique of the system's proposed solutions to automatically update the decision tree driving the interface generation process. The proposed greedy update algorithm suffered from overfitting and bootstrapping problems.

While there has been little prior work on adapting interface generation systems to people's preferences, there has been extensive prior work in the general area of preference elicitation. Below I discuss some relevant work on interaction techniques for eliciting user

preference responses, as well as machine learning approaches for inferring a person's preference model.

2.3.1 User Interactions For Eliciting Preferences

A conceptually straight forward approach to personalizing a parametrized system like SUPPLE, would be to give the user direct control over the values of all parameters. However, SUPPLE's behavior is governed by several dozens of parameters, whose values reflect complex interface design trade-offs, making the manual parameter tuning a tedious and error-prone process. The work of [116] further highlights the distinction between a user's fundamental objectives (e.g., an aesthetically pleasing interface) and the means of achieving them (e.g, the parameter values), warning that if the system attributes do not reflect their fundamental objectives, users will be unlikely to accurately estimate the values of those attributes.

Example critiquing [116] (sometimes referred to as "tweaking" [21]) is an alternative interaction technique that allows users to provide their feedback by proposing (or choosing) incremental improvements to solutions generated by a system. This type of interaction has been used in several systems, such as FindMe [21] and two trip planning assistants: the ATA system [81] and the work presented in [116]. Because SUPPLE supports a large set of possible user-driven customizations to the generated user interfaces, it can obtain example critiques from the user by exploiting these naturally occurring interactions. Example critiquing via SUPPLE's customization mechanisms has two limitations, however. First, it suffers from a bootstrapping problem, because it may be hard for the user to provide meaningful critiques until SUPPLE can generate interfaces close to desired. Second, because this interaction is entirely user-driven, it does not guarantee a complete coverage of the interface design space.

In contrast to user-driven example critiquing, system-driven interactions—where the system chooses what concrete examples to present to the user to request their opinion—can be designed to avoid the shortcomings of the user-driven approaches. A number of such interactions are commonly used. Recommender systems, for example, frequently ask users to rate the outcomes on a numerical scale (e.g., [94]). The Apt Decision agent [131] asks

users to rank order a small number of sample results. In yet another system [66], users train the system's activity recognition component by watching a video and annotating it with different attentional or interruptability states. During the same interaction, they are asked to assigned dollar values to the costs of being interrupted by different means in different situations. Recent research [23] suggests, however, that pairwise comparisons may be the most robust interaction for eliciting preferences. In that approach, the user is presented with two outcomes and is asked which of the two he or she prefers. This is the approach I adopted in my work as a system-driven complement to the user-driven example critiquing interaction.

2.3.2 Generating Optimal Queries

System-driven interactions, such as pairwise comparisons, require that the system generates examples that the user can provide feedback on. The challenge is to choose the sequence of examples in such a way, as to minimize the total number of interactions required before the system has enough data to build a comprehensive model of a person's preferences. Much of the past work on generating optimal queries is based on *value of information* (see, for example, [26, 15, 60]). In my case, it is computationally prohibitive because it requires hypothesizing about how each potential query will impact the presentation of future interfaces.

My query generation algorithms are based on the ideas of [139] and [125], who present heuristic approaches in the context of support vector machine training that aim to select queries that would maximally reduce the size of the remaining version space.

2.3.3 Representation And Reasoning About User Preferences

The systems described in [116] use constraints to represent user preferences. In order to accommodate inconsistent requests, these systems use constraint solvers that allow partial satisfaction of constraints, but no mechanism is provided to select among different solutions at the same level of feasibility.

As a refinement of this approach, the ATA system [81] uses weighted constraints but it is unclear where the weights come from. ATA considers solutions with the fewest violated constraints, but in most cases there is more than one such option. ATA responds by trying to select a small subset of maximally different options among those that are equally feasible.

Chajewska, et al. [25] propose treating utilities as random variables. The constraints encoding user's preferences are just a means for constructing a Bayesian model to estimate values of utilities, using the Metropolis sampling algorithm for inference. It is worth noting that other, potentially faster and more accurate methods are available for inference in hybrid Bayesian networks, for example [95], but their implementation complexity renders their application problematic for now.

Finally, [16] reasons about utilities in terms of minimizing the maximum regret while representing uncertainty about the value of utility parameters as hard intervals. Without further refinement, this approach does not allow for inconsistent responses from the user.

2.4 Adapting to Motor and Vision Abilities

Many others have recognized the benefit of creating graphical user interfaces specialized for people with unusual motor and visual abilities and a number of specialized interfaces have been manually designed. For example, EyeDraw [64] provides a convenient way for eye-tracker users to create art, while VoiceDraw [55] allows people to paint strokes with non-verbal vocalizations.

While there exist systems that address the scalability challenge of adapting *any* application to the needs of users with vision impairments (e.g., [11]), few do it for users with motor impairments. For example, Mankoff et al. [89] created a system that automatically modifies web pages for the needs of users with severely restricted motor abilities. Meanwhile, Input Adapter Tool [22] offers the possibility of modifying user interfaces of any application written in Java Swing to improve the accessibility for users with motor impairments. However, this system can generally only replace widgets with similarly-sized alternatives (e.g., text boxes with combo boxes) and cannot affect the organization of the interface or the sizes of the interactors.

Others, arguing that user interfaces need to be assembled dynamically for individual users [124], have developed components of necessary infrastructure [122, 123], but no general artifact seems to be available for evaluation at this time.

2.5 Adapting to Usage

The first rigorous study of adaptation was reported in 1985, when Greenberg and Witten [51] demonstrated a successful adaptive interface for a menu-driven application. In that study, the structure of the hierarchical menu adapted automatically over time to minimize the expected depth at which desired items would be found. In their study, as in most others, users were novices on the task and the interface, and long-term effects were not studied. This turned out to be an important factor. A later study by Trevellyan and Browne [140] demonstrated that the positive effects observed by Greenberg and Witten disappeared as the participants gained experience with the non-adaptive system and developed a memory for the location of frequently accessed items. A recently developed model [28] provides an analytical framework for anticipating what menu adaptations and under what usage patterns will result in performance benefit to the user.

A later study by Mitchell and Shneiderman [96] provided one of the first strong negative results: a (different) menu-driven interface, which adapted by reordering elements in the menu based on their relative frequencies of use. While this reordering could plausibly result in improved performance, users had reduced performance in practice, reported being disoriented by the changing nature of the interface, and expressed a strong preference for the static menus, where items were listed in the alphabetical order.

In 1994 Sears and Shneiderman demonstrated that another approach to adapting menus, called Split Menus, resulted in improved performance and user satisfaction [129]. However, the interface was adapted only once per user in a long-term study and once per session in a performance study. It is important to note that their original Split Menu design caused promoted elements to be moved rather than copied to the top of the menu; they were no longer available in their original locations. Some commercially deployed versions of Split Menus changed this behavior so that promoted items were available both in the original location and at the top of the interface.

Although a number of adaptive systems were soon developed, the next rigorous studies of adaptation were reported only recently. In 2002, McGrenere et al. reported a long-term study comparing the "out of the box" interface shipped with Microsoft Word to a very sparse one, which each user customized for his or her needs [91]. Additionally, the default interface had the Microsoft Smart Menus adaptation turned on, while the customized version had that adaptation turned off. The study showed that users appreciated having the personalized interface and that they used it more than the factory-supplied version. The authors interpreted the results as evidence against using adaptation, although there were large functionality differences between the two UI designs. Therefore, it is difficult to tell if the main effects of their study were due to the difference in the complexity of the two interfaces, or to the fact that one of them exhibited adaptive behavior.

In 2004 Findlater and McGrenere conducted a laboratory study to compare static, customizable and adaptive versions of Split Menus (using the original design where items are moved to the top location) [32]. They found that users generally preferred the customizable version to the adaptive menus. In terms of performance, adaptive menus were not faster than either of the two other conditions. From this study, they concluded that user-driven customization is a much more viable approach for personalizing UIs than system-driven automatic adaptation.

In 2005, Tsandilas and Shraefel reported on a study which, rather than evaluating particular adaptive interfaces, instead focused on the impact of accuracy of the adaptive algorithm on users' performance and satisfaction. They compared two different adaptive interfaces: the baseline, where suggested menu items were highlighted, and shrinking interface, which also reduced the font size of non-suggested elements [141]. In both cases increased predictive accuracy of the adaptive algorithm improved performance and satisfaction and the effect was strongest for the shrinking interface.

In another study, Findlater and McGrenere [33] investigated the impact of screen size and the adaptive algorithm's accuracy on user's performance, satisfaction, and awareness of user interface features. Using the Split Menu approach (but modified to copy rather than move promoted items to the top), they demonstrated that high accuracy adaptation was more beneficial on small screen devices, where users stood to gain more from the adaptation.

They also demonstrated that high accuracy adaptation resulted in the users being less aware of other functionality available in the interface.

Research on user-driven customization has shown that users often fail to customize [85, 108, 109] and when they do, they often fail to re-customize, as their work habits change [91]. Hybrid solutions have been suggested, for example [19], where an adaptive mechanism suggests the most useful customizations to the user. Meanwhile, adaptation has now been adopted in some mainstream commercial applications. For example, the Start Menu in Microsoft Windows XP^{TM} has an adaptive area that provides automatically generated shortcuts to frequently used applications, thus saving users from having to traverse one or more levels of program menus. Microsoft Office also features Smart Menus, an adaptive mechanism where infrequently used menu items are hidden from view. While no formal study results have been published on either of these interfaces, strong anecdotal evidence exists to suggest that the Start Menu causes few, if any, negative reactions while the Smart Menus in Office inspire strong reactions, both positive and negative, among different users.

Chapter 3

DECISION-THEORETIC USER INTERFACE GENERATION

A scalable way to provide each user with interfaces that are adapted to her devices, preferences, abilities, and tasks is to generate these interfaces automatically at run time. A number of researchers have identified this challenge and several solutions have been proposed, for example, [101, 107, 114, 105]. Although promising, the previous solutions do not handle device constraints in a general manner so as to accommodate the wide range of display sizes and interaction styles available in today's environments. For example, iCrafter [114] relies on hand-crafted templates, the Ubiquitous Interactor [105] always assigns the same interactors to the same data types assuming that sufficient space will be available, and the Personal Universal Controller [101] makes rough assumptions about the screen size (PDA) and cannot deal with situations when the most desirable rendition of an interface does not fit in the available area. Tools like Damask [80], TERESA [111], and Gummy [93] greatly simplify the process of designing user interfaces for several platforms at once, but even they cannot cope with the situations where device constraints cannot be anticipated in advance and where interface functionality is generated dynamically. Finally, previous systems do not address the individual preferences or abilities among users in rendering the interfaces.

With my Supple system I take a different approach—treating interface generation as an optimization problem. When asked to render an interface (specified functionally) on a specific device and for a specific user, Supple searches through the space of all the renditions that meet the device's constraints and chooses the one that minimizes the value of a *cost function*. This cost function can reflect a number of different concerns, such as the person's preferences or abilities, or the similarity of the current presentation of a user interface to those the user saw previously, perhaps even on a different device. Because the cost function can also incorporate information from usage traces, the automatically generated user interfaces naturally reflect the person's typical tasks and work style.

In this chapter, I make the following contributions:

- I precisely define of interface generation as a constrained decision-theoretic optimization problem. This general approach allows us to generate "optimal" user interfaces given a declarative description of an interface, device characteristics, available widgets, and a user- and device-specific cost function.

- I present an efficient rendering algorithm that can dynamically render an interface, typically in less than a second, even for complex interfaces.

- I present two types of cost functions: one factored and parametrized in a manner that supports efficient computation, and the other designed specifically to reflect the expected speed of use. Both types of cost functions incorporate usage traces, allowing SUPPLE to generate interfaces that reflect a person's long term usage patterns.

- I extend the cost function to incorporate the notion of presentation consistency among different variants of a user interface, even if they are generated for different devices.

- I introduce an implementation of the Split Interface approach to adaptation to user's short-term tasks and usage patterns; this approach is based on the results of my user studies reported in Chapter 6.

- I present a user-driven customization mechanism for SUPPLE, which allows users to change the presentation and organization of functionality in any SUPPLE-generated interface.

- I present aspects of SUPPLE's architecture that make it practical even on computationally-impoverished devices, such as phones and PDAs.

- I present an evaluation of the systems issues in SUPPLE (a user evaluation is presented in Chapter 5) .

3.1 Design Requirements

In keeping with the thesis of this dissertation, SUPPLE directly supports adaptation to the capabilities and limitations of devices, and to the person's long-term as well as short-term usage patterns and—as I demonstrate in subsequent chapters—it enables adaptation to the user's subjective preferences (Chapter 4), and to his or her objective motor and vision capabilities (Chapter 5).

However, in order to make future adoption possible, SUPPLE also needs to meet a number of practical requirements:

- It has to be **versatile** enough to support different types of applications, widget toolkits, and interaction styles, although I currently limit this requirement to graphical user interfaces only.

- It has to be **portable** to a variety of devices.

- It has to be **fast** enough to enable on-demand generation of personalized interfaces for on-line applications, like those accessible over the web.

- It has to be **scalable** in that it should enable a broad range of adaptations without requiring expert intervention.

- While enabling novel types of interfaces and interactions, SUPPLE needs to address both the opportunities and challenges to **usability** that it creates.

3.2 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* ($\mathcal{T}$). I describe each of these components below.

3.2.1 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, 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.

The upper part of Figure 3.1 illustrates 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 set of *interface elements*, and $\mathcal{C}_\mathcal{I}$ is a set of *interface constraints* specified by the designer.

The interface elements included in the 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 designer-specified interface constraints can, in principle, constrain any aspect of interface presentation. In practice, SUPPLE currently only supports a means for constraining multiple instances of the same type (e.g., all three lights in the Classroom interface) to be rendered identically. The customization mechanism described in Section 3.7 also uses the interface constraints to specify what concrete presentation elements should be used to render a particular part of the interface.

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 3.1. The 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 3.2, where a user 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.

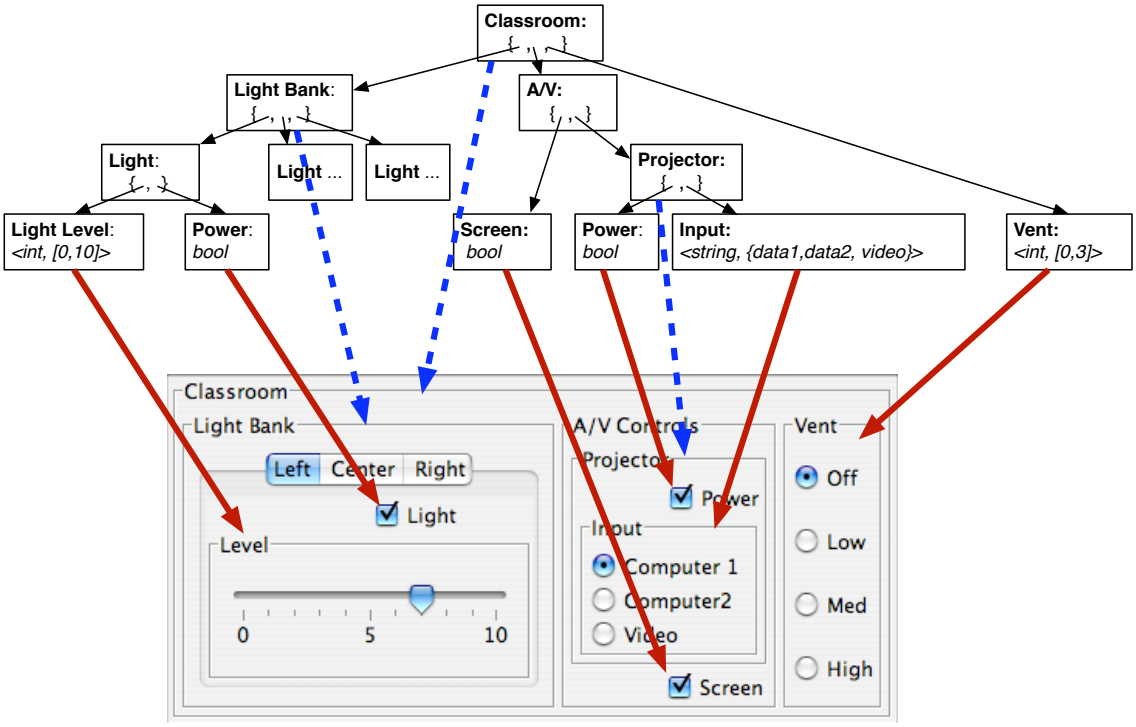


Figure 3.1: SUPPLE uses an algorithm that makes discrete assignments of widgets to the elements of the functional interface specification. This figure illustrates a functional specification and a sample concrete user interface for an application controlling a small set of devices in a classroom. The 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.

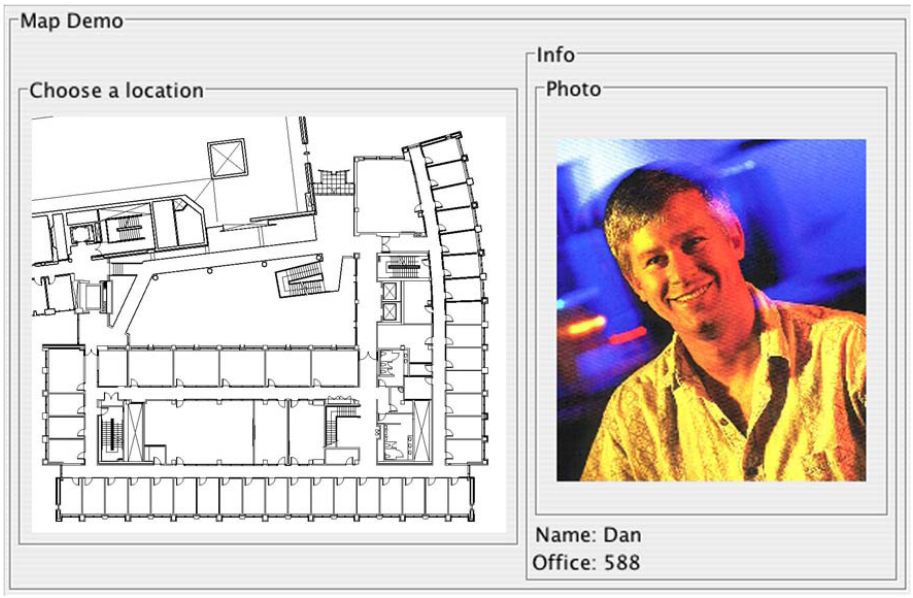


Figure 3.2: An interface utilizing images and clickable maps.

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 3.1 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: $\langle \tau, \mathcal{C}_\tau \rangle$ denotes a constrained type, where τ is any primitive or container type and $\mathcal{C}_\tau$ 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 3.3a 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

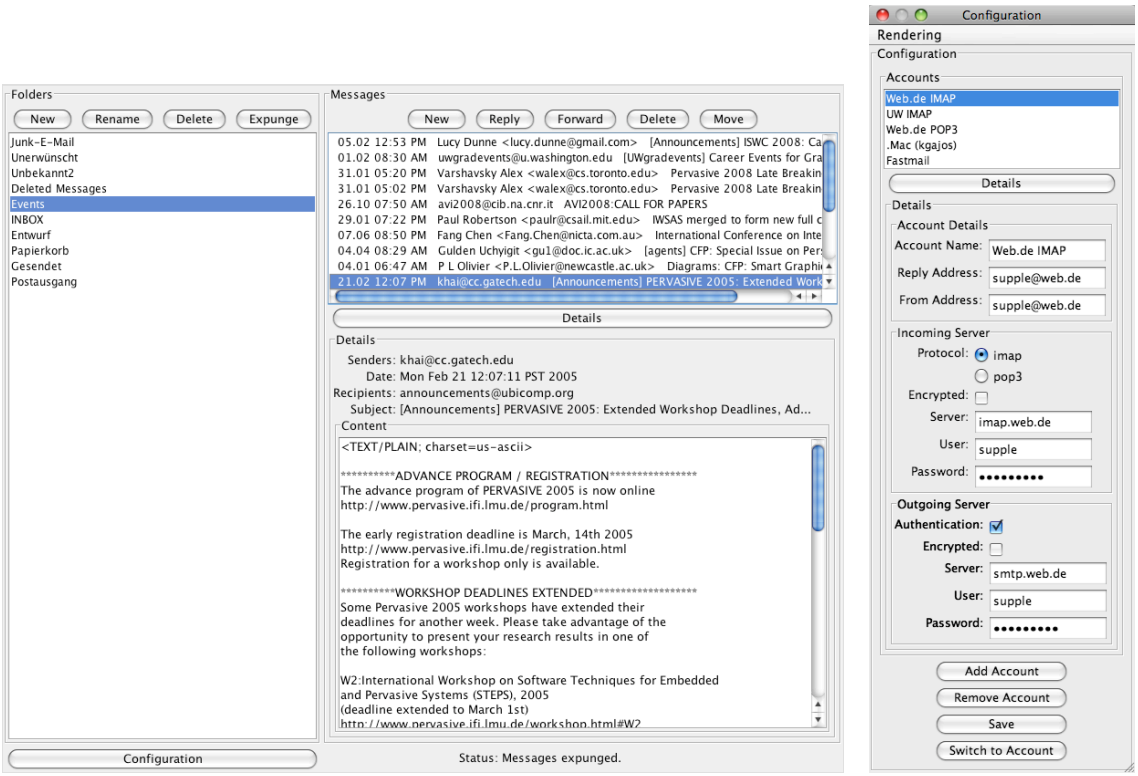


Figure 3.3: An email client that uses SUPPLE to render its user interface. (a) The main view. (b) The configuration pane.

container types. For example, consider the list of available email accounts in the email example of Figure 3.3b. 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 3.3b. When space is 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, what is the expected size of the domain of possible choices, and so on.

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 other interface description languages, such as those used in the Personal Universal Controller [101] or TERESA [111] 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 3.4. 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 items do not. To alleviate this problem, SUPPLE allows the elements of a container of type τ to be a subtype τ' of τ.[1] In such situations, if space permits, SUPPLE renders all the attributes of the common ancestor type τ statically, next to the choice element (Figure 3.4a). Any time a specialized object is selected by the user, another

[1] 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.

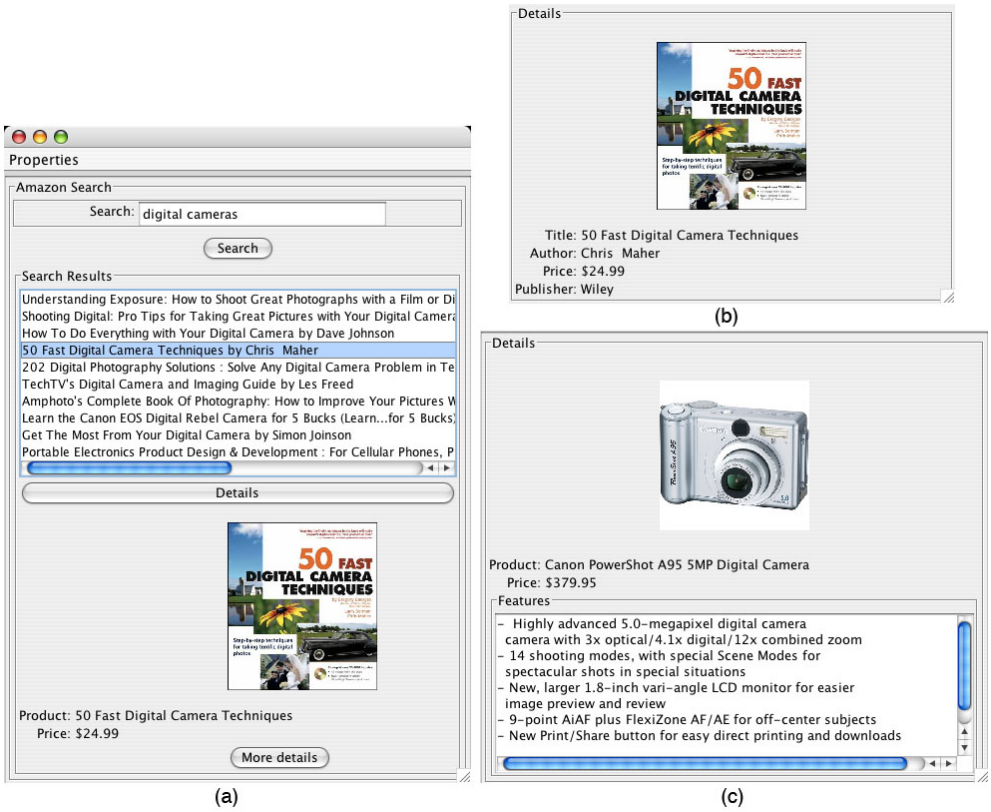


Figure 3.4: A simple client for Amazon Web Services. (a) Search results with a pane showing properties of a selected object. Only those properties which are common to all items are shown there, but the "More Details" button brings up a specialized view for each item. (b) Detailed view for a book. (c) Detailed view for a digital camera.

button is highlighted, alerting the user that more detailed information is available, which can be displayed in a separate window as shown in Figures 3.4b and 3.4c.

Vectors: elements of type $vector(\langle \tau, \mathcal{C}_\tau \rangle)$ 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 $\mathcal{C}_\tau$ define the set of values of type τ that can be selected. For example, the list of emails in the email client (Figure 3.3a) 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 as $\tau_1 \mapsto \tau_2$, where τ_1 stands for the type of the object containing parameters of the action, while 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*. The "Reply" button in the rendering of the email client interface in Figure 3.3a is represented as an action. Its return type is Message and its parameter type is *null*; although it operates on the current message, so do the "Forward," "Delete," and "Move" actions, so the current message is implemented as a global variable in this context. The Search in the Amazon browser example in Figure 3.4a is an action with string as a parameter type and—because it does not cause a new interface element to be displayed—it has a *null* return type.

3.2.2 Device Capabilities and Constraints

I model a display-based device as a tuple:

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

where $\mathcal{W}$ is the set of available user interface widgets on that device, and $\mathcal{C}_\mathcal{D}$ denotes a set of device-specific constraints. Below I describe these elements in detail.

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

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

3.2.3 Modeling Users with Traces

Different parts of the interface may be rendered with different salience depending on their relative importance. Instead of relying on explicit annotations by the designer or the user, SUPPLE relies on usage traces, which can correspond either to actual or to 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 user trace, $\mathcal{T}$, is a set of *trails* where, following [146], 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). The important property of a trail is that if two subsequent events refer to different interface elements, this sequence of events indicates a transition made by the user. I assume that a trail ends when the interface is closed or otherwise reset. I define a trail T as a sequence of *events* u_i, each of which is a tuple $\langle e_i, v_{old_i}, v_{new_i} \rangle$. Here e_i is the interface element manipulated and v_{old_i} and v_{new_i} refer to the old and new value this element assumed (if appropriate). It is further assumed that $u_0 = \langle root, -, - \rangle$, where *root* stands for the root element in the functional specification tree.

As the trace information accumulates, it can be used to re-render the interface adapting it to the needs of a particular user. This, of course, has to be done very carefully because frequent changes to the interface can be distracting. Also, because the format of a trace is independent of a particular rendering, the information gathered on one device can be used to create a custom rendering when the user chooses to access the application from a different device. In note that in some cases use of different devices may be correlated with different contexts of use (for example, a user 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 ensures that our algorithm works 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.

3.3 Interface Generation as Optimization

The goal is to render each interface element with a concrete widget, as illustrated earlier in Figure 3.1. 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 $\mathcal{C}_\mathcal{I}$ and $\mathcal{C}_\mathcal{D}$. Of course, there may be many legal renderings. Therefore, in order to find the best one, SUPPLE relies on a *costs function* **$**, 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 (Chapter 4) or expected speed of use (Chapter 5), and it can incorporate additional concerns, such as similarity to previously seen renderings of a user interface, even if those renderings were generated for other devices (Section 3.5).

I thus define an *interface rendering problem* formally as a tuple $\langle \mathcal{I}, \mathcal{D}, \mathcal{T}, \$ \rangle$, where $\mathcal{I}$ is a description of the interface, $\mathcal{D}$ is a device model specifying the size constraints and available widgets, $\mathcal{T}$ is the user trace, and **$** is the cost function. R is a *solution* to a rendering problem if R is a legal rendering with minimum cost—I 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.

3.3.1 Optimization Algorithm

Conceptually, SUPPLE enumerates all possible ways of laying out the interface and chooses the one which minimizes the user's expected cost of interaction. Efficiency is obtained by using branch and bound search [79, 97] and a novel admissible heuristic to explore the space of candidate renderings; full propagation of size constraints and forward checking of interface constraints are used to maximize the pruning effect of violated constraints.

Table 3.1 shows our algorithm for solving the interface rendering problem. The branch-and-bound search is guaranteed to find an optimal solution. The *remainingCostEstimate*

function referenced in line two of the *optimumSearch* procedure is an admissible heuristic for pruning partial solutions whose cost is guaranteed to exceed that of the best solution found so far. This function looks at all unassigned variables (i.e., nodes in the functional specification that do not have a single concrete widget assigned to them yet) and adds the costs of the best available widgets for each of these variables (ignoring constraints). Note that while this is a very natural formulation of the admissible heuristic for this problem, it critically relies on a factoring of the cost function that allows it to be computed as a sum of costs of individual variables. Finding such a factoring is the topic of the next section, while Section 3.4 introduces an alternative admissible heuristic for a case where such a factoring cannot be found.

The search is directed by 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. I have experimented with three variable ordering heuristics: *bottom-up* chooses the variable corresponding to the deepest widget in the interface specification tree (Figure 3.1), 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 [121]; the idea is always to focus on the most constrained variable, that is, the one with the fewest possible values remaining.

I 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. For example, suppose a (large) slider widget was chosen for the vent speed; constraint propagation might immediately realize that there was no way to fit a widget for light intensity. Furthermore, it allows SUPPLE to make more

Table 3.1: Branch and bound optimization applied to the interface generation problem. The variables (*vars*) represent widget-to-element assignments and are passed by reference. The *constraints* variable contains both the interface and device constraints.

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

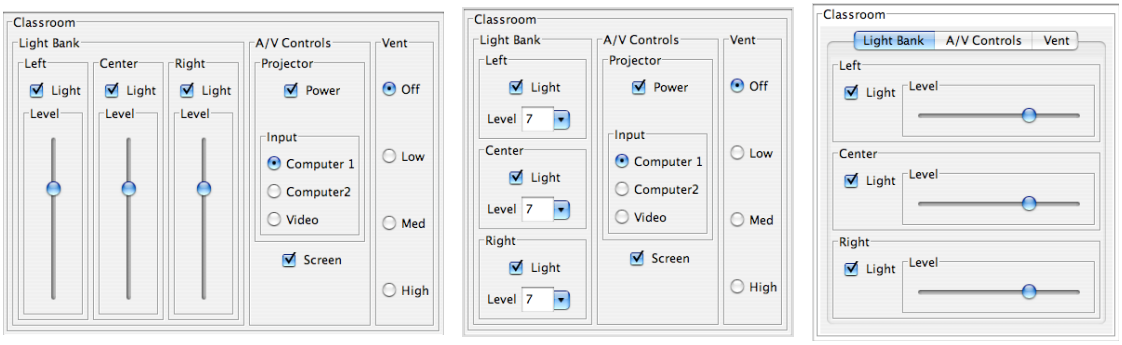


Figure 3.5: SUPPLE optimally uses the available space and robustly degrades the quality of the rendered interface if presented with a device with a smaller screen size. This figure shows three renderings of a classroom controller on three devices with progressively narrower screens.

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 [121]. 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 only 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 3.9.4) shows that these optimizations are indeed very effective.

The above algorithm is inherently discrete, while some aspects of a user interface are better modeled with continuous values. For example, the length of a list widget for showing search results in the Amazon search interface (Figure 3.4) can vary reasonably from a handful to 40 entries. SUPPLE addresses the problem by providing a few discretized approximations (e.g., lists of 5, 10, 15 elements).

Compared to previous rule-based approaches, optimization is relatively robust, flexibly handling 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 3.5 illustrates how SUPPLE robustly degrades the quality of the generated user interfaces as it is presented with devices with progressively narrower screens.

3.3.2 Factoring The Cost Function

In earlier parts of this chapter, I have alluded to several requirements for the form the cost function should take. Here, I recount them formally:

1. The cost function should take into account the 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. As argued in previous section, to enable efficient computation of the admissible heuristic that the optimization algorithm relies on (Table 3.1, line 2), the cost function needs to be factorable such that it can be computed as a sum of costs over all elements in the functional specification. With such a cost function, the cost of a left light element in the Classroom interface in the left part of Figure 3.5, for example, would be calculated as a sum of the cost of using a slider for the light level, the cost of using a checkbox for the on/off switch, and the cost of using the vertical layout for combining the two.

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.

In this section I derive a cost function that satisfies these three requirements. I start by defining $\$$ to be of the form:

$$\$(R, \mathcal{T}) \equiv \sum_{T \in \mathcal{T}} \sum_{i=1}^{|T|-1} N(R, e_{i-1}, e_i) + \mathcal{M}(R(e_i)) \tag{3.1}$$

where N is an estimate of the effort of navigating between widgets corresponding to the subsequent interface elements referenced in a trail. $\mathcal{M}$ 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 3.1 satisfies the first of the three requirements but requires re-analyzing the entire user trace each time a new cost estimate is necessary and fails to satisfy the remaining two requirements.

To address those limitations, I 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 3.6). If we consider a container widget w representing an interface element e, the three transitions are *sibling switching* (sw), when user manipulates two elements that belong to two different children of e; *leaving* (lv), when a child of e is manipulated followed by an element that is not a child of e; and *entering* (ent), which is the opposite of leaving. For different types of container widgets, these three transitions are predicted to increase user effort in different ways. For example, suppose that e represents a tab pane widget; $\mathcal{N}(\mathsf{sw}, e)$ denotes the cost of switching between its children and would be high, because this maneuver always requires clicking on a tab pane. Leaving a tab widget requires no extra interactions with the tab. Entering a tab pane usually requires extra effort unless the tab the user is about to access has been previously selected. In case of a pop-up window, both entering and leaving require extra effort (click required to pop-up the window on entry, another click required to dismiss it) but no extra effort is required for switching between children if they are rendered side by side.

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

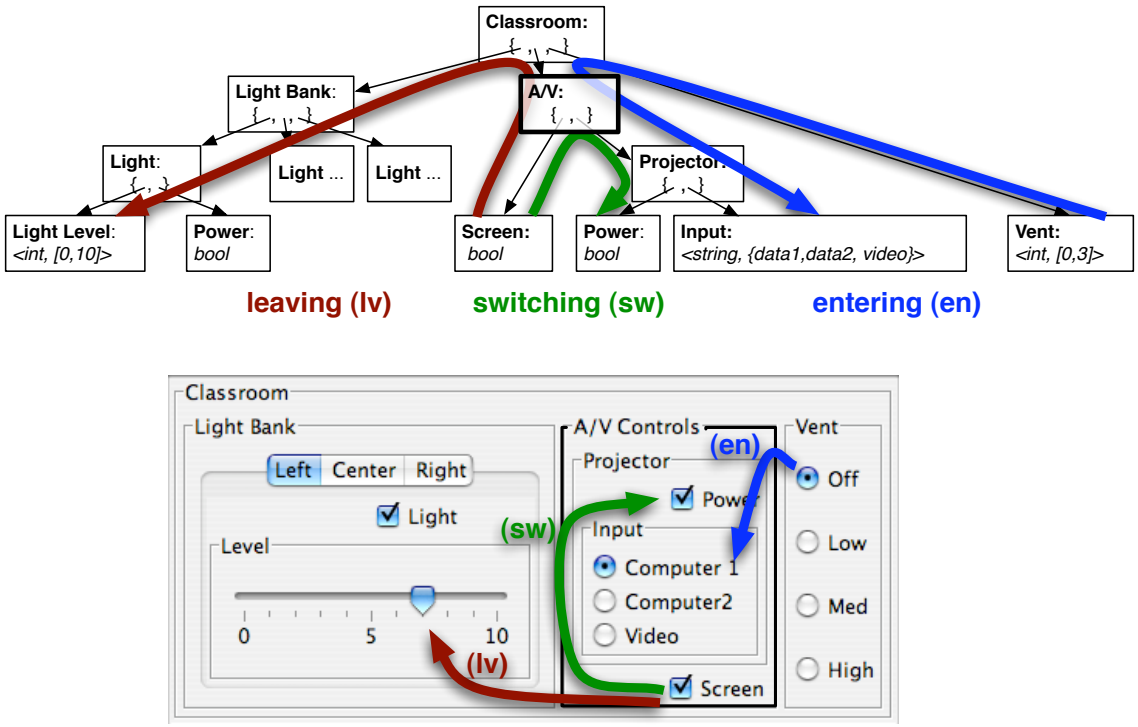


Figure 3.6: Three types of transitions between elements of a user interface with respect to the A/V interior node: *leaving* (lv), when a child of A/V node is manipulated followed by an element that is not a its child; *sibling switching* (sw), when user manipulates two elements that belong to two different children of the node; and *entering* (ent), which is the opposite of leaving.

interface, one can unambiguously determine the path between any two elements in the interface. I denote the path between elements e_i and e_j to be $p(e_i, e_j) \equiv \langle e_i, e_{k_1}, e_{k_2}, \ldots, e_{k_n}, e_j \rangle$. I thus choose the navigation cost function, N, from Equation 3.1 to be of the form:

$$N(R, e_{i-1}, e_i) = \sum_{k=1}^{|p(e_{i-1},e_i)|-2} \begin{cases} \mathcal{N}(\mathsf{sw}, R(e_k)) & \textit{if } \text{child}(e_k, e_{k-1}) \wedge \text{child}(e_k, e_{k+1}) \\ \mathcal{N}(\mathsf{lv}, R(e_k)) & \textit{if } \text{child}(e_k, e_{k-1}) \wedge \text{child}(e_{k+1}, e_k) \\ \mathcal{N}(\mathsf{ent}, R(e_k)) & \textit{if } \text{child}(e_{k-1}, e_k) \end{cases} \quad (3.2)$$

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 in the interactions recorded in the user trace. I 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, I may state the component cost of an interface *element*, $R(e)$, as:

$$\begin{aligned} \$(R(e), \mathcal{T}) = \quad & \#_{\mathsf{sw}}(\mathcal{T}, e) \times \mathcal{N}(\mathsf{sw}, R(e)) \\ + \quad & \#_{\mathsf{ent}}(\mathcal{T}, e) \times \mathcal{N}(\mathsf{ent}, R(e)) \\ + \quad & \#_{\mathsf{lv}}(\mathcal{T}, e) \times \mathcal{N}(\mathsf{lv}, R(e)) \\ + \quad & \#(\mathcal{T}, e) \times \mathcal{M}(R(e)) \end{aligned} \tag{3.3}$$

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

$$\$(R(\mathcal{S}_f), \mathcal{T}) = \sum_{e \in \mathcal{S}_f} \$(R(e), \mathcal{T}) \tag{3.4}$$

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, I 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

$$f_{slider\_for\_number}(R(e),\mathcal{T}) = \#(\mathcal{T},e) \times \begin{cases} 1 & \textit{if type of } e = \text{number} \wedge R(e) = \text{slider} \\ 0 & \text{otherwise} \end{cases} \tag{3.5}$$

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:

$$f_{list\_undersize}(R(e),\mathcal{T}) = \#(\mathcal{T},e) \times \begin{cases} \frac{\text{number of choices}}{\text{list size}} & \textit{if } R(e) = \text{list} \\ & \wedge \text{ number of choices} > \text{list size} \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

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 [61].

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

$$f_{tab\_switch}(R(e),\mathcal{T}) = \#_{\mathsf{sw}}(\mathcal{T},e) \times \begin{cases} 1 & \textit{if } R(e) = \text{tab pane} \\ 0 & \text{otherwise} \end{cases} \tag{3.7}$$

This factor will return the number of switch transitions for container elements rendered as tab panes in a concrete user interface and 0 in all other cases.

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, I can rewrite Equation 3.3 as follows:

$$\$(R(e),\mathcal{T}) = \sum_{k=1}^{K} u_k f_k(R(e),\mathcal{T}) \tag{3.8}$$

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 3.4 and 3.8, the final formulation of the cost function used by SUPPLE is as follows:

$$\$(R(\mathcal{S}_f), \mathcal{T}) = \sum_{e \in \mathcal{S}_f} \sum_{k=1}^{K} u_k f_k(R(e), \mathcal{T}) \tag{3.9}$$

The current implementation of SUPPLE for desktop user interfaces relies on nearly 50 factors.

The manual choice of the appropriate weights is a difficult and error-prone process. In Chapter 4, I introduce the ARNAULD system, which automatically learns the right values of these weights based on a small number of preference statements expressed by the user over concrete examples of user interfaces. The results reported in Section 5.7.2 indicate that this set of factors is expressive enough to capture most of the *subjective* aesthetic and usability concerns of desktop computer users.

3.4 Optimizing GUIs for 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 [128].

However, using this metric results in a significantly harder optimization problem and involves an additional continuous parameter—the size of the clickable elements of the widgets—because the size of clickable elements is one of the critical parameters influencing the users' speed of navigating to and clicking on user interface elements.

In this section, I show how to extend the general optimization framework introduced earlier in this chapter to generate user interfaces, which are optimized for a user's quantitative *performance* given a predictive model of how fast a person can perform basic user interface operations. In Chapter 5, I develop such a model together with a set of diagnostic tasks for eliciting a person's motor abilities.

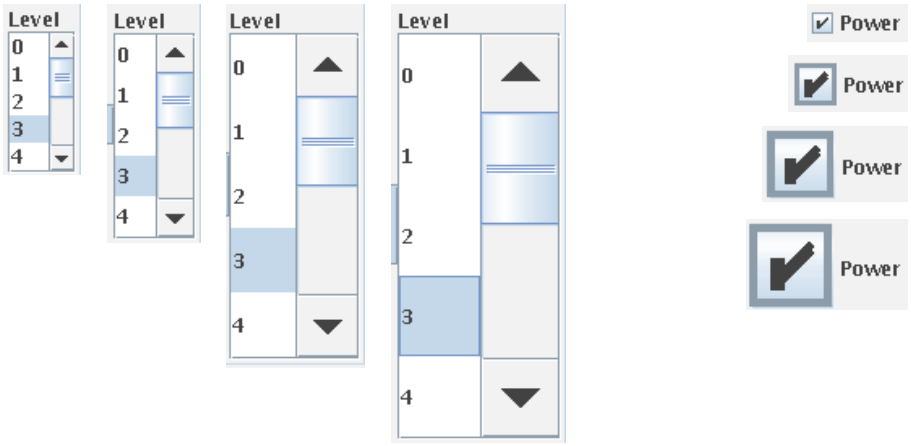


Figure 3.7: Presentation of a list widget and a checkbox widget for different values of the minimum target size constraint s.

3.4.1 Reformulating the Cost Function

The cost function factoring introduced in Equation 3.9 in Section 3.3.2 has the important properties that the cost of each widget can be computed independently of others, and that the cost associated with navigating the interface is separated from the usability of individual interactors. These two properties were leveraged to develop an efficient admissible heuristic for the branch-and-bound search algorithm for finding the optimal rendering.

A more complex cost function is required, however, to optimize the interfaces for a user's expected movement time, EMT:

$$\begin{aligned} \$(R(\mathcal{S}_f, s), \mathcal{T}) &= EMT(R(\mathcal{S}_f, s), \mathcal{T}) \\ &= EMT_{nav}(R(\mathcal{S}_f, s), \mathcal{T}) + \textstyle\sum_{e \in \mathcal{S}_f} EMT_{manip}(R(e, s), \mathcal{T}) \end{aligned} \tag{3.10}$$

Here, EMT_{nav} is the expected time to navigate the interface, EMT_{manip} is the expected time to manipulate a widget (0 for layout widgets) and s is the minimum target size; that is, the minimum size for all control elements of a widget that can be manipulated with a mouse pointer (see Figure 3.7 for two examples of how widgets are drawn depending on the

different values of s). There are two important differences between this formulation of the cost function and that formulated in earlier sections:

1. The lengths and the target sizes for the movements between the widgets, and thus the EMT_{nav}, cannot be computed until all interactors and layout widgets have been chosen. This is problematic because it makes it hard to estimate the minimal cost of renderings consistent with a partial assignment, blocking the use of branch-and-bound, and ruining search efficiency.

2. Both the expected time to manipulate a widget and the expected time to move between widgets depend on the new continuous parameter, s, transforming a discrete optimization problem into a harder, hybrid (discrete/continuous) one.

The rest of this section explains how I confront these problems. I begin, however, with a brief description of the process for computing EMT_{manip} for primitive widgets.

Computing EMT_{manip}

Many widgets can be operated in different ways 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 visible elements. Scrolling may be operated 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, I 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 [87]. Although more accurate models for two-dimensional pointing have been developed for typical mouse users [2, 52], those models are unlikely to be equally appropriate for unusual devices, interaction techniques and users with motor impairments (i.e., the intended targets for this extension to SUPPLE), and I 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.

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 that 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, I proceed as follows. First, for each unassigned leaf node, e, I compute a rectangular area which is guaranteed to be covered by all of the widgets which are compatible with e; that is, I compute the minimum width of all compatible widgets and separately find the minimum height, as illustrated below.

One may now propagate these bounds upwards to form bounds on 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

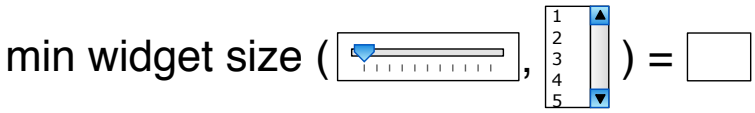


Figure 3.8: Computing minimum of widget sizes for primitive widgets.

node has not yet been assigned a specific layout, then I again independently compute the minimum of the possible widths and the possible lengths.

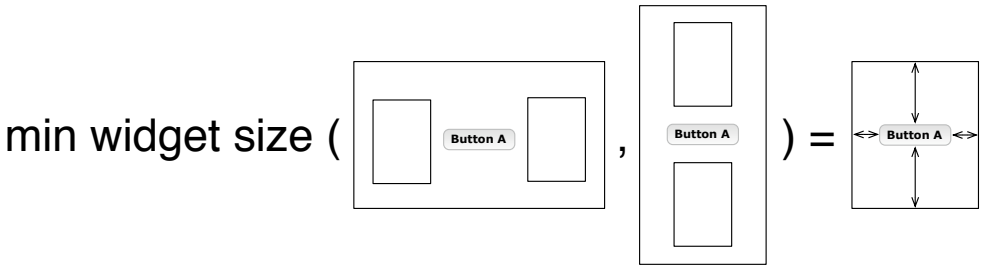


Figure 3.9: Computing minimum of widget sizes for container widgets. The result is not only the minimum dimensions of the bounding box but also the minimum distance between any of the enclosed elements and the bounding box.

Note, however, that in this case for each element contained within a layout element (like the Button A in Figure 3.9), our estimate also provides the minimum distance from the edges of the layout element to the contained element. As a result, one can compute the most *compact* possible layout for an interface and thus the shortest possible distance between any pair of elements, as illustrated below:

To provide a lower bound on the time to move between elements e_s and e_t, I 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, e_t. 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.

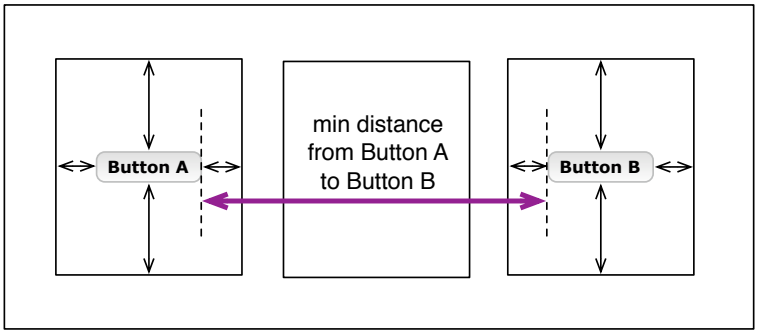


Figure 3.10: Computing minimum distance between two elements in a layout.

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.

My evaluation (Section 3.9.5) shows that this lower bound on EMT_{nav} resulted in dramatic improvements to the algorithm performance.

3.4.2 Optimizing the Minimum Target Size

In contrast to my original formulation of the optimization problem, however, in this formulation SUPPLE must also optimize s, the minimum target size, which is a continuous parameter. Figure 3.11 shows how the cost (the EMT) of the best GUI varies as the minimum target size ranges between 0 and 100 pixels. Because these curves include numerous local minima, one can't 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. 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. Because this approach allows SUPPLE to continue to exploit branch-and-bound, search is very fast in practice for a great many values can be pruned quickly.

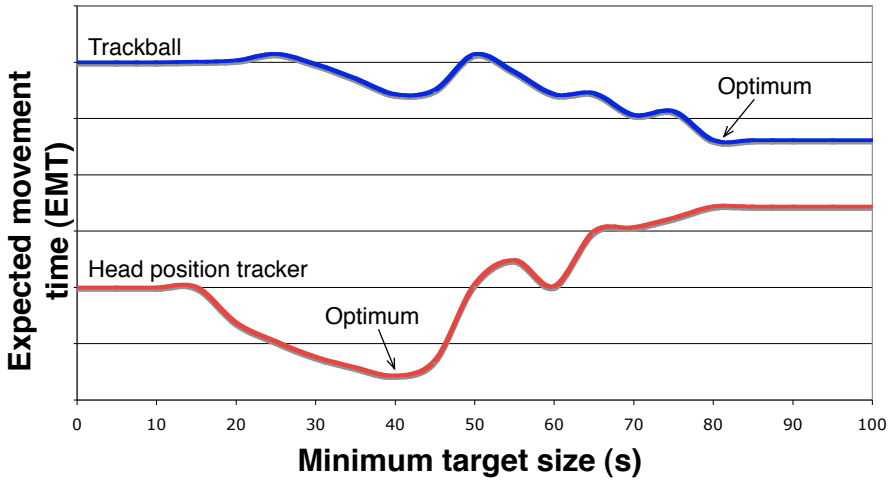
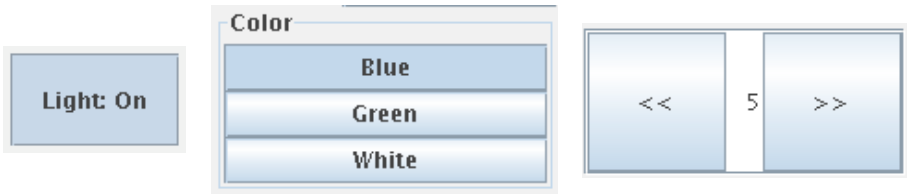


Figure 3.11: The estimated movement times (EMT) of the optimal GUIs generated with different values of the minimum target size parameter for two participants. Y-axis corresponds to the EMT but the curves were shifted to fit on one graph so no values are shown.

3.4.3 Extending a GUI Toolkit to Allow For Variable Target Size

We changed the presentation of standard widgets (to allow for resizing of interaction targets, icons, and fonts) by re-implementing parts of Java Swing's Metal Look and Feel[2]. I have further implemented three additional simple widgets for SUPPLE to use as alternatives to a checkbox, a set of radio buttons, and a spinner, respectively:



[2]Jing Jing Long contributed to the implementation of this extension [46].

These alternative widgets are appropriate for situations where user's dexterity is impaired, either due to the context of use (e.g., using a laster pointer to control the mouse cursor from a large distance [99]), or due to a health condition.

3.5 Consistency Across Interfaces Generated 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 [120]. 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, I present an extension to SUPPLE's cost function that allows it to reflect dissimilarities in visual appearance and organization between pairs of interfaces[3]. The objective is that if an interface that was once rendered on one device (for example, a desktop computer) 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}:

$$\$(R(\mathcal{S}_f), \mathcal{T}, R_{ref}(\mathcal{S}_f)) = \$(R(\mathcal{S}_f), \mathcal{T}) + \alpha_s \Delta(R(\mathcal{S}_f), R_{ref}(\mathcal{S}_f)) \tag{3.11}$$

In Equation 3.11, $\mathcal{T}$ 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 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.

---

[3] Anthony Wu contributed to the design and implementation of SUPPLE's consistency mechanisms [43].

As with the base cost function, 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 is computed by iterating over all elements e of the functional specification $\mathcal{S}_f$ of an interface and sum over all factors:

$$\Delta(R(\mathcal{S}_f), R_{ref}(\mathcal{S}_f)) = \sum_{e \in \mathcal{S}_f} \sum_{k=1}^{K} u_k f_k(R(e), R_{ref}(e)) \tag{3.12}$$

Thus the dissimilarity function decomposes in the same way as the original cost function described in Section 3.3.2 allowing for incremental computation, which maximizes the effectiveness of the optimization algorithm.

3.5.1 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 pairs 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) the widget uses to convey its value; for example, the slider uses the position, list uses text and position, checkbox uses toggle.

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

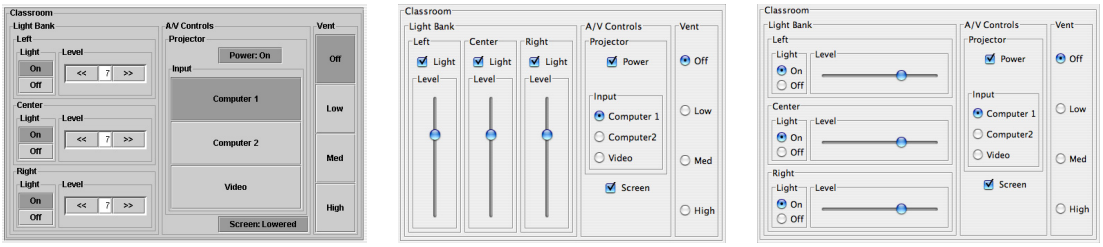
Primary manipulation method {point, drag, text entry} – the primary way of interacting with the widget.

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 3.12 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.



(a) (b) (c)

Figure 3.12: 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.

3.6 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 my experiments reported in Chapter 6, I implemented the Split Interface approach 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 previous implementations of this general approach, which could only adapt contents of menu items or toolbar buttons, 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 copy that functionality to the adaptive part of the main print dialog box (Figure 3.13).

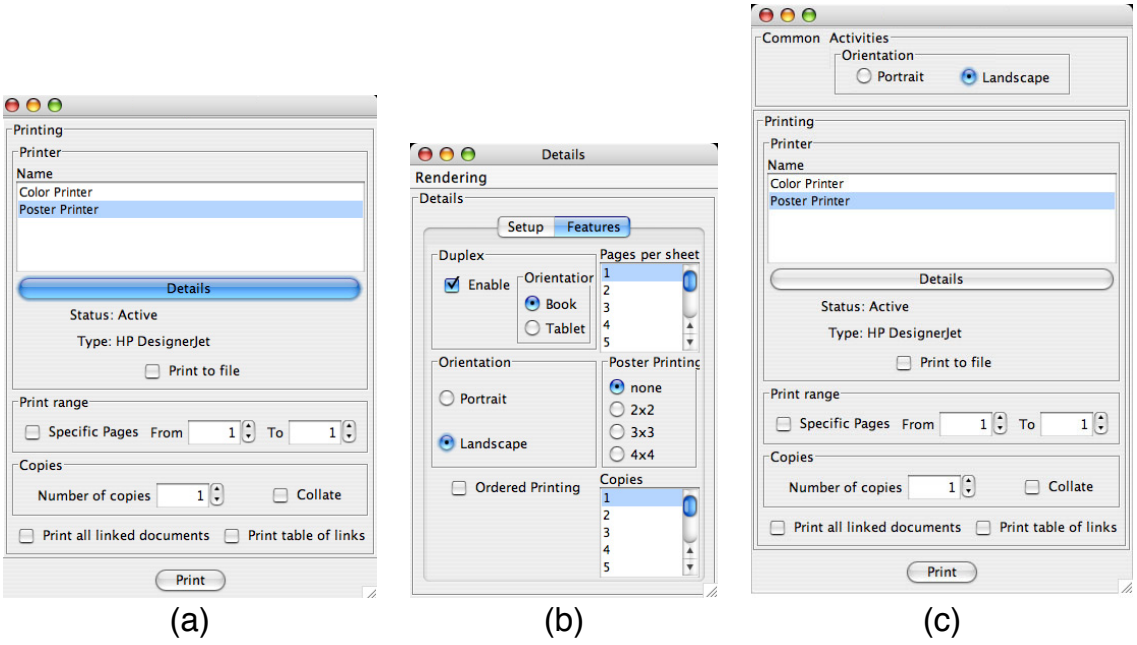


Figure 3.13: In the original print dialog box it takes four mouse clicks to select landscape printing: (a) details button, (b) features tab, landscape value and then a click to dismiss the pop-up window. (c) shows the interface after automatically adaptation by SUPPLE given frequent user manipulation 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.

3.7 User-driven Customization

In earlier parts of this chapter, I introduced two system-driven approaches to adapting user interfaces in SUPPLE. In this section, I introduce a complementary user-driven *customization* mechanism[4].

Just as with traditional user interfaces, users may want the ability 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

[4]The initial implementation of SUPPLE's customization facilities was done by Raphael Hoffmann [40, 39].

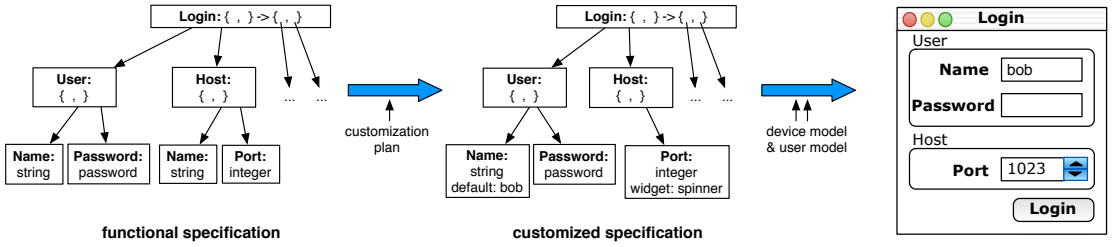


Figure 3.14: SUPPLE's customization architecture. The user's customization actions are recorded in a *customization plan*. The next time the interface is rendered (possibly in a differently sized window or on a different device) the plan is used to transform the functional specification into a *customized specification* which is then rendered using decision-theoretic optimization as before.

satisfaction and performance when used to create custom simplified versions of the interface that are streamlined for the user's individual tasks and habits [91, 92].

SUPPLE includes a comprehensive *customization* facility that allows a designer or 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 3.14, customizations are recorded as a *customization plan* and they are represented as modifications to the functional specification rather than 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.

This approach allows customizations performed on one device to be reproduced on other devices, with possible exception of customizations to the interface presentation if 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 dependancy 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 3.15 offers an example of a user interface where both presentation and organization of the interface had been customized. Another more in-depth example is also discussed in Section 3.9.3.

3.8 Implementation

SUPPLE is written in Java and is designed to integrate easily with existing application code, especially with applications whose state is maintained in Java Beans — in such cases SUPPLE automatically maintains two-way consistency between the interface and application states. Communication between SUPPLE's implementation components is illustrated in Figure 3.16. User interfaces are specified by creating UI objects for each property or method to be exposed through the interface. When asked to generate a concrete user interface, SUPPLE proceeds in the following way:

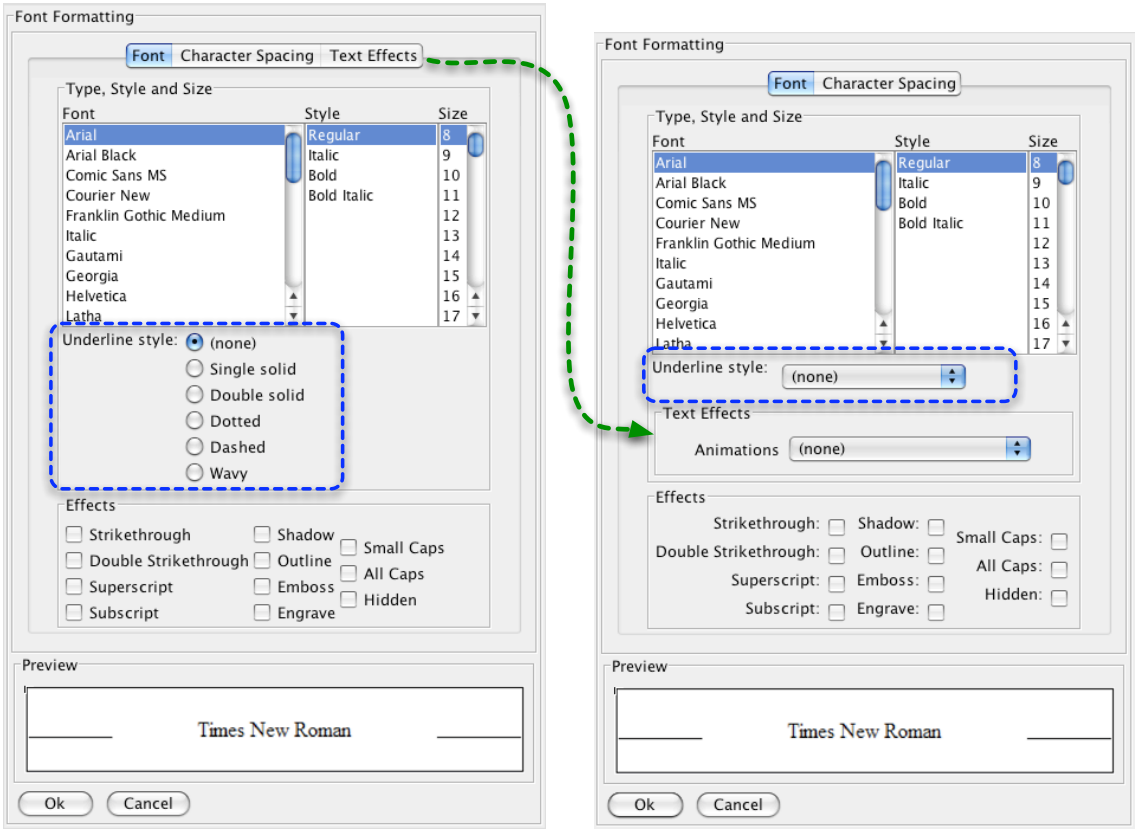


Figure 3.15: An illustration of the customization mechanism: (left) an interface for a font formatting dialog generated automatically by SUPPLE; (right) the same interface customized by the user: the Text Effects section was moved from a separate tab to the section with main font properties, and the presentation of Underlying Style element was changed from radio buttons to a combo box.

1. The device model is engaged to generate a set of candidate widget *proxies*, serializable objects that are capable of generating a concrete widget, for each element of the interface model.

2. Using combinatorial search, SUPPLE generates the best assignment of widget proxies to interface model elements and establishes the connections between them. If neces-

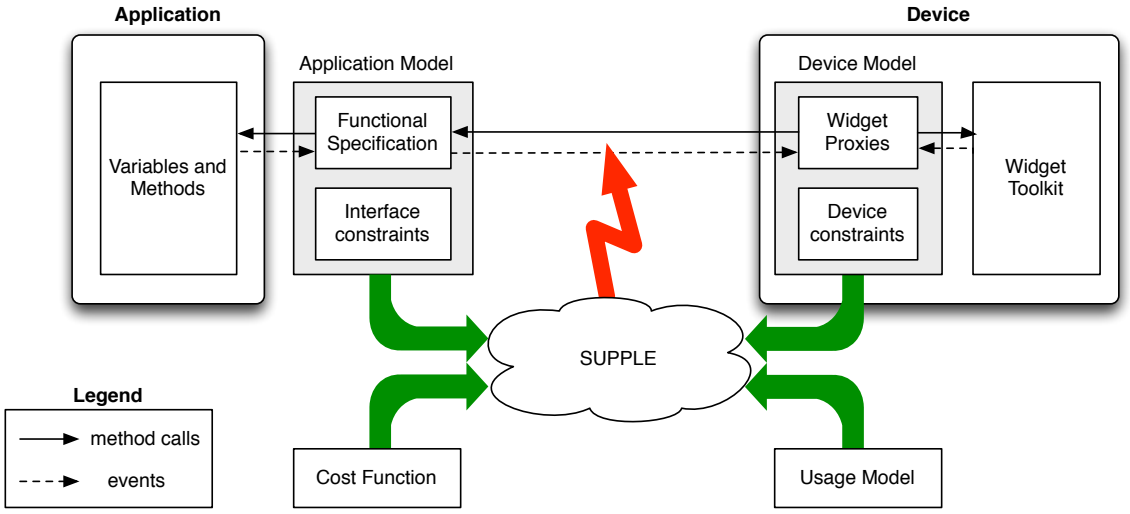


Figure 3.16: SUPPLE's implementation: The interface model exposes the state variables and methods of the application that should become accessible through the interface. The widget proxies generated by the device model are assigned to interface model elements by SUPPLE's optimization algorithm, which additionally relies on a cost function and a usage model in generating a concrete user interface. When delivered to the final device, the Widget Proxies are triggered to generate the concrete user interface and subsequently mediate communication with the platform specific widgets.

sary, the proxies may now be serialized for caching or transport to computationally-impoverished devices.

3. The widget proxies are triggered to generate a concrete user interface on the final display device.

For most applications, new windows or dialog boxes need to be displayed at run-time. SUPPLE renders them dynamically as needed.

3.8.1 Handling Computationally-Impoverished Devices

In some situations, users will want to access applications on their desktop computers (PCs). In others, they will want to use their desktop computers to access applications running

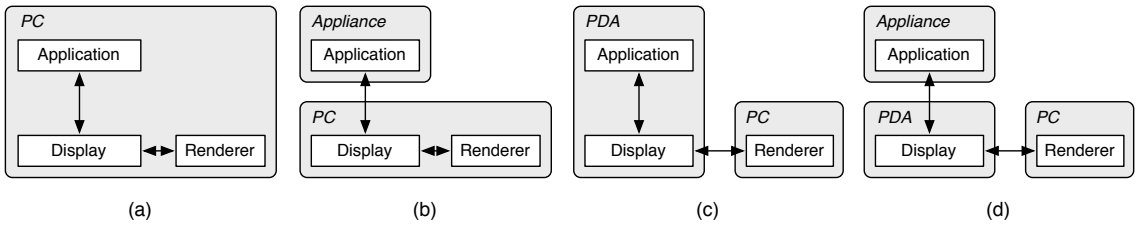


Figure 3.17: SUPPLE allows the application, renderer server, and interface to run on different devices; the following modes of operation are common: (a) An application running on a PC is displayed on the same machine — the user interface is rendered locally. (b) A remote application is displayed on a PC — the user interface is rendered on the PC. (c) An application running on a PDA is displayed on that same PDA — a remote server may be used for faster rendering of an interface which is then cached. (d) A remote application is displayed on a PDA — again a remote server may be used to quickly fill the cache with the required interfaces.

on a remote server or appliance. Finally, mobile users may want to access either local or remote applications using computationally-impoverished devices such as PDAs and cell-phones. These scenarios require SUPPLE to present interfaces on devices other than those executing application logic[5].

Furthermore, the measurements show that SUPPLE optimization-based rendering can take up to 40 seconds to render an interface on a PDA such as a Dell Axim v50x. To address this challenge, SUPPLE can also utilize a remote rendering service (I refer to it as the *renderer server*) to accelerate the rendering process whenever network connectivity is available. Figure 3.17 illustrates different modes of operation supported by SUPPLE. In order to enable disconnected operation and to save power, aggressive caching of rendering results is also supported. In the remainder of this section, I summarize SUPPLE's distributed architecture.

Being able to render interfaces for remote applications is a common feature of today's interface generation systems, like the previously mentioned Personal Universal Controller [100]

[5]The support for distributed operation was implemented by David Christianson [39].

and the Ubiquitous Interactor [105]. However, the computational complexity of SUPPLE's decision-theoretic rendering algorithm creates unique challenges for the distribution of this system.

SUPPLE supports distribution of the application and the interface through a distributed framework, based on Java RMI. SUPPLE automatically choses local or remote bindings depending on the configuration, and the application programmer does not need to be aware that distributed operation is involved.

As a bootstrapping measure, SUPPLE includes a local discovery mechanism based on Multicast DNS (the base protocol for Apple's Bonjour) so that display devices can easily detect available applications and renderer servers in their environments. Developers are free to replace it with whatever local or global discovery mechanism is most appropriate for their deployment environment.

3.9 Evaluation of Systems Issues

In this section I examine SUPPLE's technical capabilities and limitations while in Chapter 5, I report on a user study.

3.9.1 Versatility

I demonstrate SUPPLE's versatility by exhibiting the range of different types of interfaces it has generated. Earlier in the chapter, I presented an interactive map-based interface (Figure 3.2), a fully functional email client (Figure 3.3), an interface to Amazon web services (Figure 3.4), a dialog box for font formatting (Figure 3.15), and an interface for controlling lighting, ventilation and audio-visual equipment in a classroom (Figure 3.5). Figure 3.18 shows this last interface rendered for 5 different platforms: a touch panel, an HTML browser, a PDA, a desktop computer and a WAP cell phone[6]. Figure 3.19 shows a user interface for controlling a stereo rendered on a PDA and on a desktop computer. Finally, Figure 3.22 shows Microsoft Ribbon re-implemented in SUPPLE.

[6]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 [39].

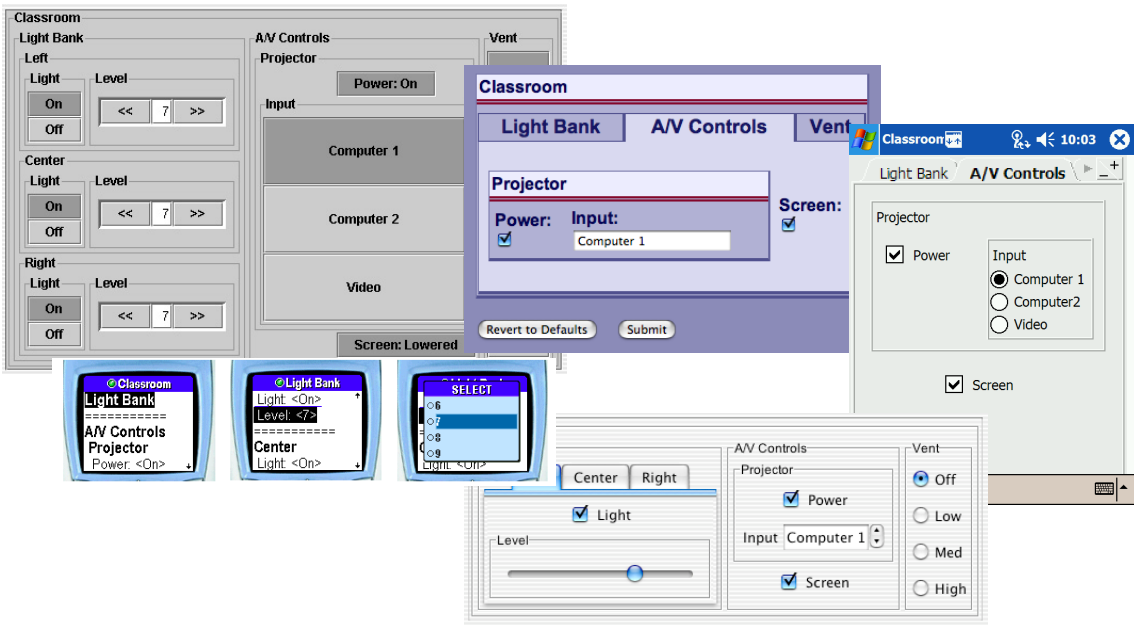


Figure 3.18: An interface for the classroom controller application rendered automatically by SUPPLE for a touch panel, an HTML browser, a PDA, a desktop computer, and a WAP phone.

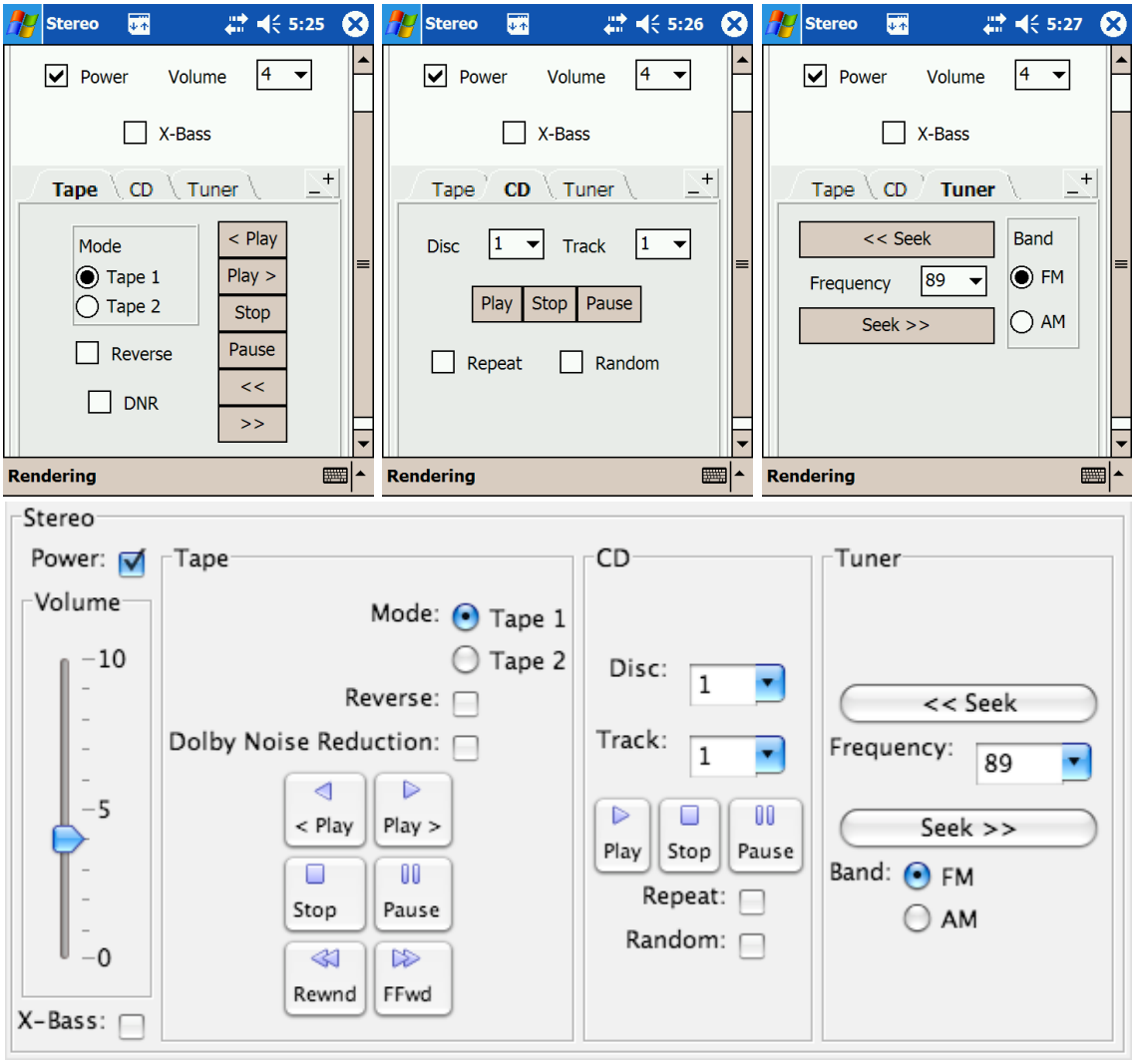


Figure 3.19: A user interface for controlling a stereo rendered automatically by SUPPLE for a PDA (top) and on a desktop computer (bottom).

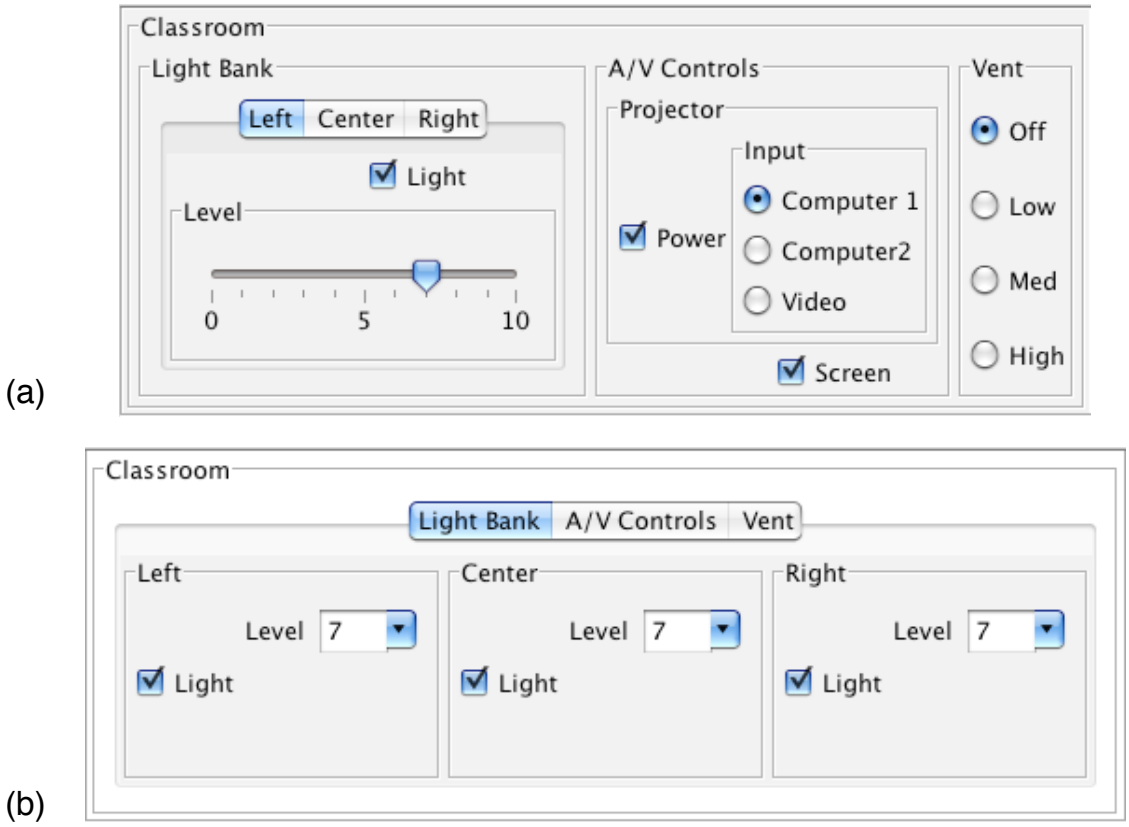


Figure 3.20: The classroom interface rendered for a small screen size: (a) with an empty user trace (b) with a trace reflecting frequent transitions between individual light controls.

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).

3.9.2 Adapting To Long-Term Usage Patterns

Both formulations of the cost function introduced in this section incorporate usage statistics from a usage model. These statistics impact how SUPPLE generates user interfaces. For example, Figure 3.20 shows two versions of the classroom interface rendered under the same size constraint. The two interfaces were generated in response to two different usage models. The rendition in Figure 3.20(a) was based on a usage trace that represented uniform usage

of all the features, and the one in Figure 3.20(b) 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 and only a portion of the available space, makes it easier to navigate between individual light controls than the first one.

3.9.3 User-driven Customization

Microsoft Ribbon (Figure 3.21) is an interface innovation introduced in Microsoft Office 2007 as a replacement for menus and toolbars. One of it's 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 3.22a shows a fragment of the Ribbon re-implemented in SUPPLE, while Figure 3.22b shows that same fragment adapted to fit in a narrower window.

The size adaptation of MS 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 in the case of toolbars from earlier versions of MS Office, in Ribbon there is no mechanism to allow moving, copying, adding or deleting buttons, panels or other functionality.

SUPPLE's automatic interface generation algorithm, which takes size as one of the input constraints, automatically provides the size adaptations (Figure 3.22b). More importantly, however, SUPPLE's customization mechanisms allow people to add new panels to the SUPPLE version of the Ribbon as well as move, copy, and delete functionality. The customized Ribbon can be naturally adapted to different size constraints by SUPPLE (Figure 3.22c). In this case, automatically generated and adapted interactions can improve users' sense of control compared to the manually created solution.

3.9.4 System Performance

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

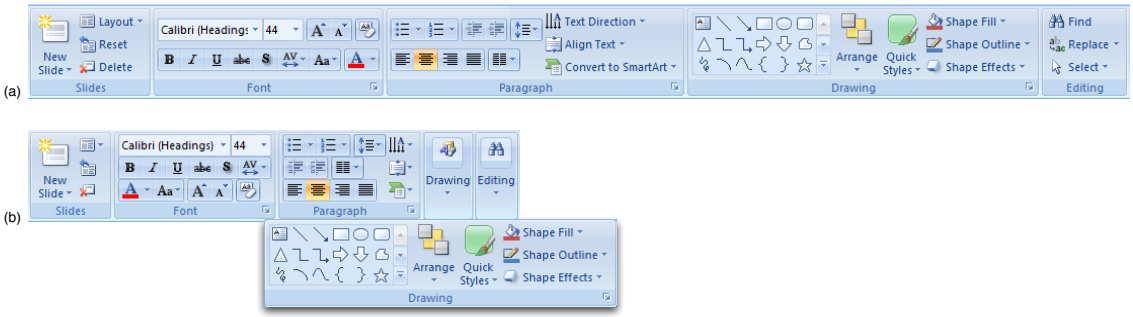


Figure 3.21: A fragment of Microsoft Ribbon (a) presented in a wide window; (b) the same Ribbon fragment adapted to a narrower window: some functionality is now contained in pop-up windows.

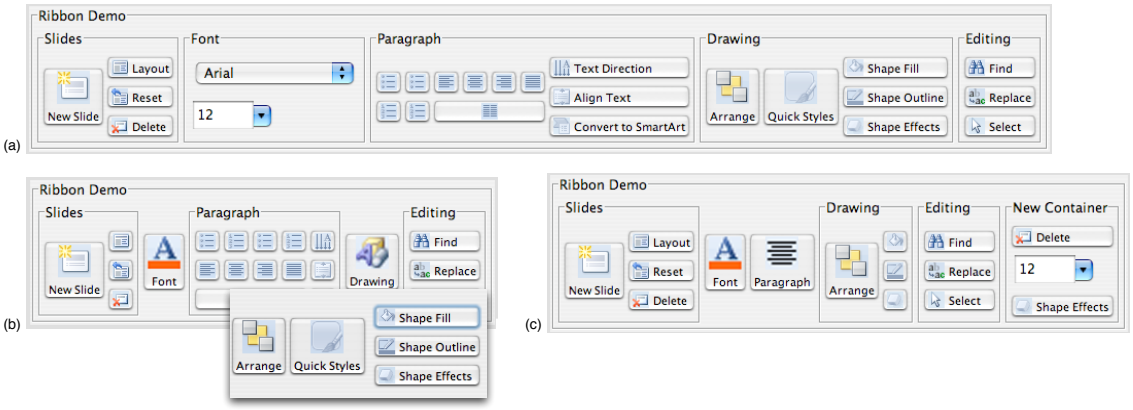


Figure 3.22: A fragment of the Microsoft Ribbon re-implemented in SUPPLE (a) rendered for a wide window; (b) SUPPLE automatically provides the size adaptations, which are manually designed in the original version of the MS Ribbon; (c) unlike the manually designed Ribbon, the SUPPLE version allows users 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.

The computational problem that SUPPLE solves to generate user interfaces is that of constrained combinatorial optimization. This is a computationally very hard problem—exponential in the number of the specification elements in the worst case—but in practice most instances of such problems are tractable with just a small number of instances being substantially harder to solve. This is frequently referred to as the *phase transition* phenomenon [115, 57, 56]. 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 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. For some problem spaces the location of the phase transition can be predicted analytically [149]. The space of user interface generation problems, however, is highly discontinuous and therefore hard to investigate analytically.

Variable Ordering Heuristics and the Parallel Algorithm

In this section, I investigate empirically both the average and the worst-case performance of my algorithm using the factored version of the cost function described in Section 3.3.2. I start by investigating the properties of the three variable ordering heuristics considered (described in Section 3.3.1): bottom-up, top-down, and minimum remaining values (MRV). To examine a representative cross-section of the problem space for each interface considered, I pick two extreme screen size constraints: one so large that a greedy approach to generating a user interface will succeed. The second just small enough that no interface can be generated for it. I interpolate at 100 intervals between these two extremes for a total of 101 screen sizes, and for each size I run the optimization algorithm, collecting the following measures:

- 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.

Figure 3.23 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 3.5), a print dialog box (introduced in Chapter 5, see Figure 5.8), and a stereo controller interface (Figure 3.19).

This figure illustrates the fact that the phase transition phenomenon is taking place. It also illustrates another important phenomenon: the MRV and bottom-up heuristics tend to exhibit the worst performance in slightly different parts of the problem space. This is an important observation because it suggests 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, I implemented two variants of a *parallel algorithm*. The first, which will be referred to as Parallel-2, runs two searches concurrently driven by two variable ordering heuristics experienced phase transition behavior in different parts of the problem space; these were 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 I expected to see the benefit of the non-overlapping phase transition regions. 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 3.1). Therefore, I 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 3.24 shows the performance of the two parallel algorithms on the three example interfaces introduced in Figure 3.23. In each case, the best performing variant from Figure 3.23 is also shown for comparison. Note the different scales used for the base line and parallel algorithms.

The average case performance in all cases remained the same as that of the single search but, as expected, the worst-case performance improved dramatically: by an order of

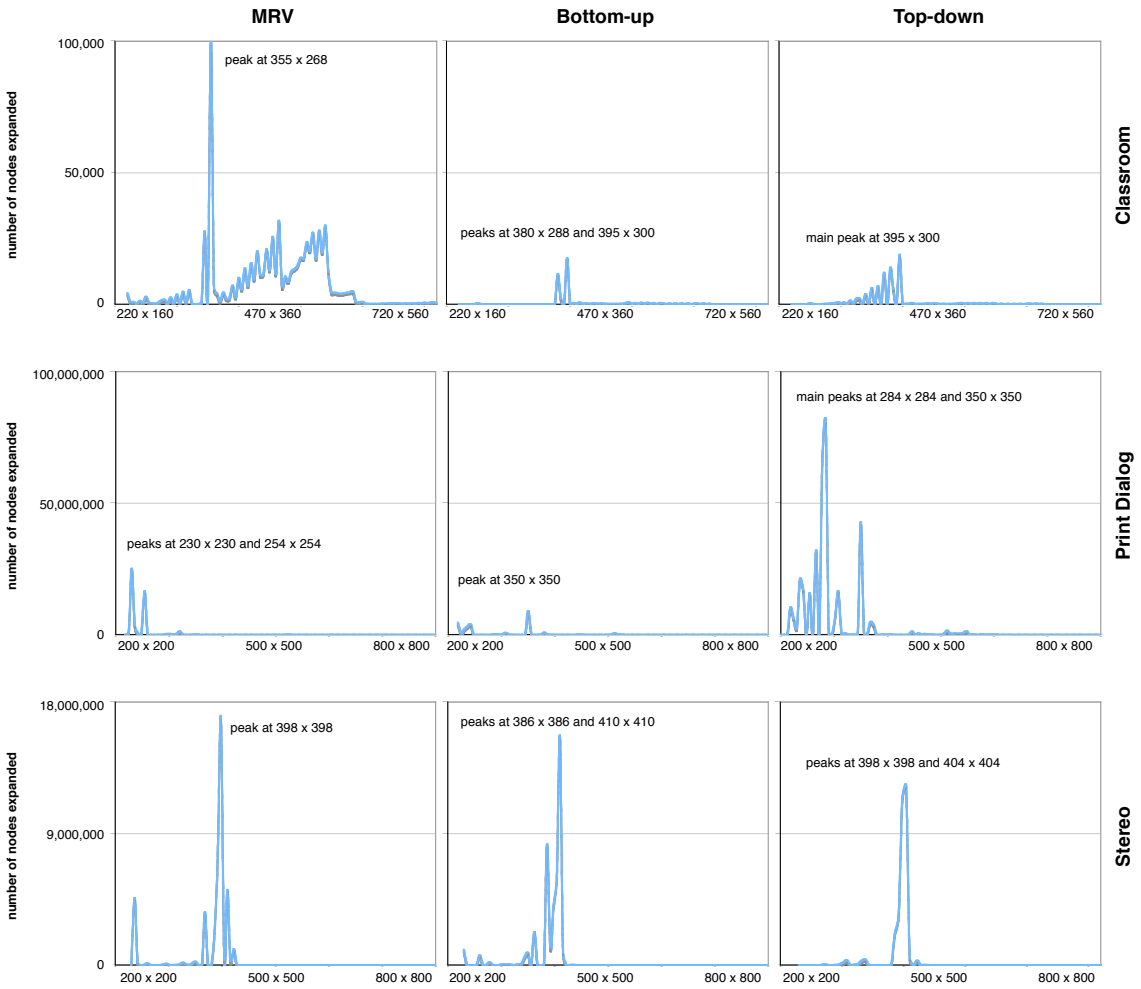


Figure 3.23: Algorithm performance (measured in number of nodes considered by the search algorithm) for three different user interfaces studied systematically over 101 size constraints for three different variable ordering heuristics: minimum remaining values (MRV), bottom-up and top-down.

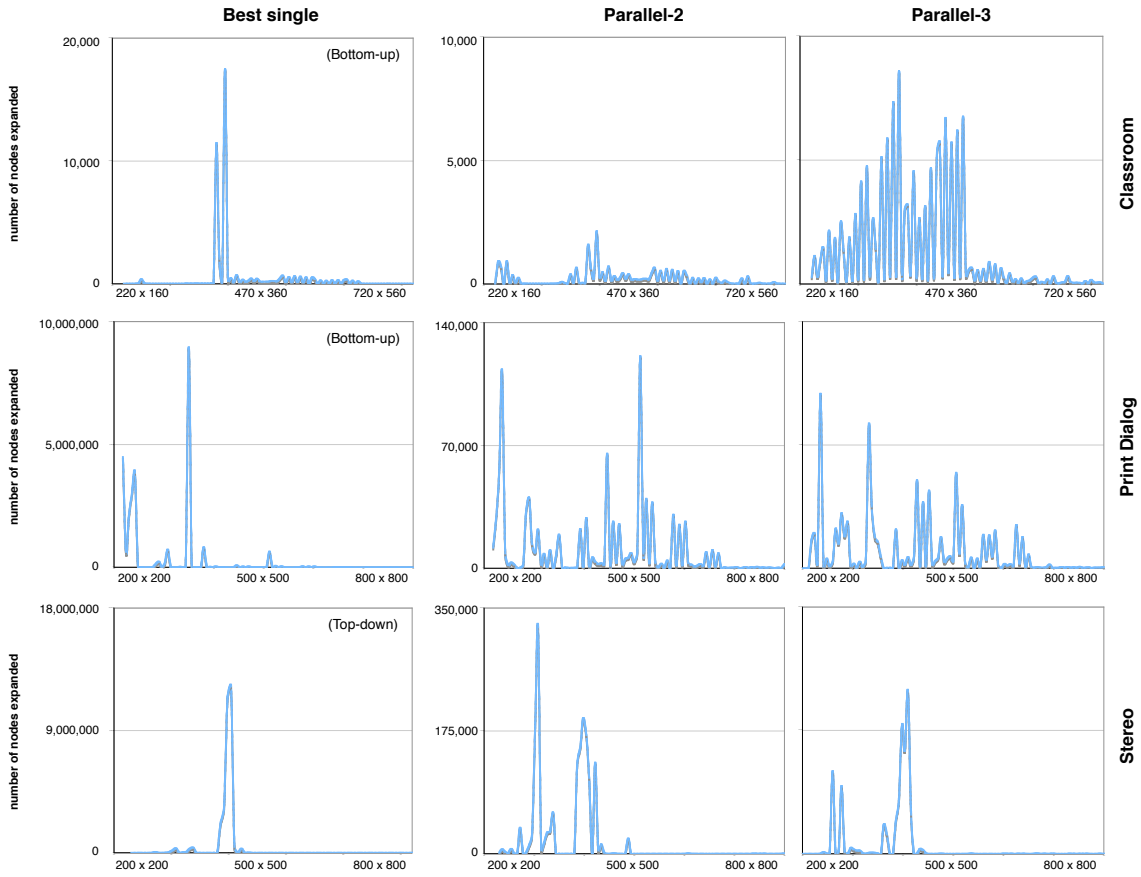


Figure 3.24: The performance (measured in number of nodes considered by the search algorithm) of the two parallel algorithms compared to the best-performing single-threaded variant from Figure 3.23. Results are presented for three different user interfaces studied systematically over 101 size constraints. Note different scales on y-axes for the parallel and single-threaded algorithms.

Table 3.2: 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.

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

magnitude in the case of the classroom interface and by *nearly two orders of magnitude* for the print dialog and the stereo interface.

Scalability

Next, I investigate how my approach scales with the complexity of the interfaces it generates. Table 3.2 summarizes the results.

I evaluated the two parallel algorithms with 11 different user interfaces, a number of which are used as examples throughout this chapter. For each interface, I first report the number of elements in the functional specification excluding those for which rendering is constrained through the same rendering constraint, and the number of the possible interface renderings that the algorithm will consider. For each interface, I conducted the experiments in the manner described earlier: measuring the performance at 101 different points throughout the problem space. I measured both the number of nodes explored and the total time taken to produce the best solution. I report both the average case values (the median across all trials) and the worst-case numbers.

Note that the number of possible interfaces considered spans 15 orders of magnitude from 216 for the map interface to the 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

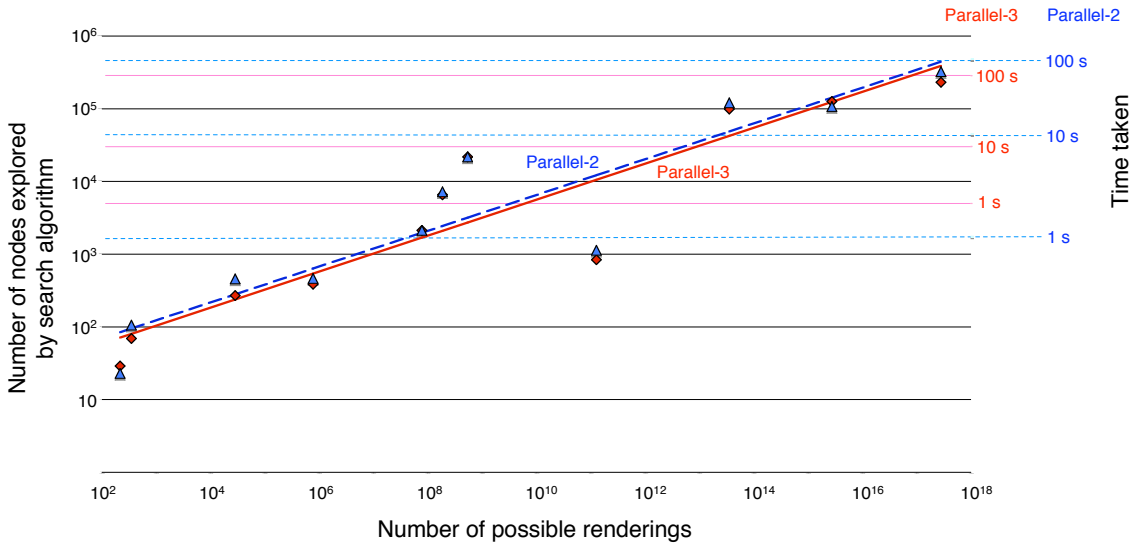


Figure 3.25: The worst-case performance of the two parallel algorithms with respect to the complexity of the user interface. The x-axis shows the number of possible user interfaces and the y-axis shows both the maximum number of nodes explored by the search algorithm (left) and the actual time taken (right). Note that the relationship between the number of nodes expanded and the time taken is slightly different for the two algorithms; hence two separate scales are provided.

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 than Parallel-3, in the average case. In the worst case, 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 change predictably with the interface complexity. 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 nodes expanded and the number of possible interfaces, n, is $22.52 \times n^{0.246}$ ($R^2 = 0.97$) and for Parallel-3 it

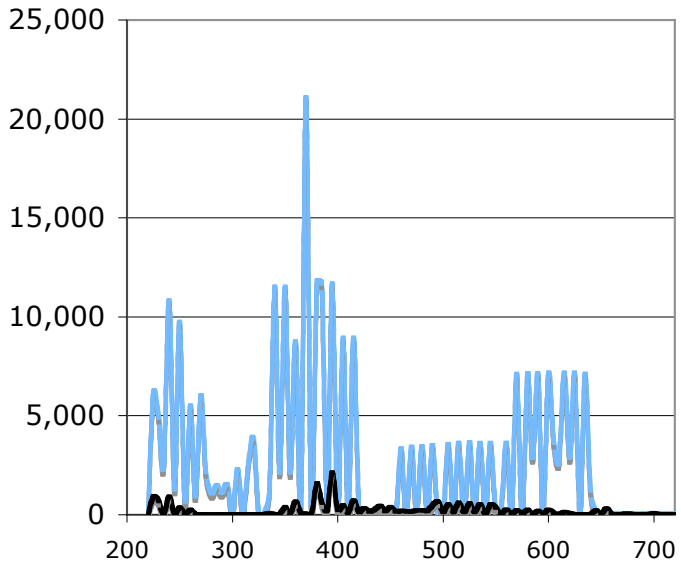


Figure 3.26: The worst-case performance of the parallel-2 algorithm with only forward checking (a limited version of constraint propagation) with respect to the complexity of the user interface. The x-axis shows the number of possible user interfaces and the y-axis shows the maximum number of nodes explored 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).

is $18.83 \times n^{0.247}$ ($R^2 = 0.94$). Figure 3.25 illustrates these relationships. Of course, the exponents smaller than 1 mean that the performance scales sub-linearly, specifically as a root of the number of possible interfaces.

Importance of Constraint Propagation

Unsurprisingly, constraint propagation has a significant impact on the algorithm's performance. Figure 3.26 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

Table 3.3: Lines of code used to construct and manage the user interfaces for several of the applications presented throughout this chapter.

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

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.

3.9.5 Performance When Optimizing For Expected Movement Time

In Section 3.4, I introduced an alternative cost function that can reflect a person's expected time to complete a set of typical tasks with a user interface. The structure of that function was incompatible with the original admissible heuristic introduced to guide the branch and bound search.

For interfaces illustrated in Figures 5.1 and 5.5, SUPPLE needed between between 3.6 seconds and 20.6 minutes to compute the user interfaces expected to minimize the total movement time. 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.

I 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.

3.9.6 Model Complexity

Comparisons of the code quantity or complexity among different approaches are often controversial. Yet, I feel it is useful to report on the amount of code[7] devoted to the description and management of the user interface for the examples reported in this chapter. These numbers are reported in Table 3.3.

3.9.7 Portability

In order to assess how SUPPLE meets the portability requirement, I have measured the system's performance on a PDA[8]. For the mid-complexity classroom interface, SUPPLE needed 29 seconds to generate the interface while performing the computation entirely on the PDA. For the much more complex stereo interface this time was 40 seconds. These times were shortened dramatically to 2.5 and 3.1 seconds, respectively, when a remote renderer running on a desktop computer was used over the network. These results demonstrate that PDA users can also experience fast interface generation times if they have network access to a remote renderer. Even disconnected PDA users are not prevented from using SUPPLE but they may have to endure a substantial wait (e.g., up to 40s) the first time any given interface is rendered. Future renderings are instantaneous because of the caching mechanism.

I have also measured the sizes of the messages that need to be exchanged in order to invoke a remote renderer. The functional specification sizes vary from 4.8kB to 11.5kB while the rendered solutions are all smaller than 3.6kB. These sizes make a renderer server an option, not only on WiFi networks, but also on slower Bluetooth or 3G cellphone connections.

3.10 Summary

In this chapter, I described SUPPLE, a system which automatically generates user interfaces given a functional user interface specification, a model of the capabilities and limitations of the device, a usage model reflecting how the interface will be used, and a cost function.

[7]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.

[8]Dell Axim x50v with an Intel 624 MHz Xscale processor and 64MB of RAM purchased in 2004.

SUPPLE naturally generates user interfaces adapted to different devices as well as different long-term usage patterns. As a complement to automatic generation and adaptation, SUPPLE also provides an extensive user-driven customization mechanism, which allows any SUPPLE-generated user interface to be substantially modified in terms of the presentation of the individual pieces of functionality and the overall organization.

SUPPLE's optimization algorithm can generate user interfaces in under a second in most cases, provided the cost function is expressed in a particular parametrized form. I have also introduced an alternative cost function formulation, which can reflect user's motor capabilities but which results in lower system performance. I assert that the style of the user interfaces generated by SUPPLE can be entirely determined by the appropriate parameterization of the cost functions. This offers a potential for personalizing the interface generation process.

The remaining challenge is to develop mechanisms that will allow end users to perform this personalization without involving expert help. Such mechanisms would allow my approach to scale to the vast range of individual devices, abilities, and preferences that exist in the world because they would remove the bottleneck of scarce expertise. The next two chapters introduce two such personalization mechanisms.

Chapter 4

ADAPTING TO PREFERENCES

The previous chapter described my SUPPLE system, which uses optimization to generate concrete user interfaces from declarative specifications of a user interface, the target device, and the expected usage model. The particular style of user interfaces generated by SUPPLE is governed by a cost function. In Section 3.3.2, I introduced a particular form of a cost function, parametrized by a set of 50 weights. Because these weights express complex trade-offs in user interface design, choosing them manually is tedious and error-prone. In order to accomplish my goal of making SUPPLE easily personalizable by non-experts, I need a fast and intuitive method for specifying those weights.

This challenge is not unique to SUPPLE. Recent years have revealed a trend towards increasing use of optimization as a method for automatically designing aspects of an interface's interaction with the user. In most cases, this optimization may be thought of as *decision-theoretic* — the objective is to minimize the expected cost of a user's interactions or (equivalently) to maximize the user's expected utility. For example, the BUSYBODY system [68] mediates incoming notifications with a decision-theoretic model of the expected cost of an interruption, which is computed in terms of the user's activity history, location, time of day, number of companions, and conversational status. The LINEDRIVE system [3], generates graphical representations of driving directions by using optimization to find the optimal balance between readability and fidelity to the original shapes, directions and lengths of the individual road segments. The RIA system [152, 153] chooses the best answer to a user query by optimizing a cost function which offsets content quality with quantity and other factors. And there are many other examples [81, 65, 67, 131, 8, 37, 38].

While decision-theoretic optimization provides a powerful, flexible, and principled approach for these systems, the quality of the resulting solution is completely dependent on the accuracy of the underlying cost (utility) function. While domain-specific learning tech-

niques have been used occasionally, most practitioners parameterize the cost function and then engage in a laborious and unreliable process of hand-tuning.

This chapter presents the fully implemented interactive ARNAULD system[1] which uses a combination of two complementary interaction techniques to solicit user feedback on concrete user interfaces or their parts. It thus frees the humans from having to reason about numerous unintuitive parameters, probabilities, or monetary values of different tradeoffs and moves the discourse into the space of concrete outcomes. In this chapter, after stating my guiding design requirements, I make the following contributions:

- I identify two types of interactions by which users can conveniently provide feedback regarding their preferences: *example critiquing* acquires implicit feedback from certain types of user actions within the interface, and *active elicitation* explicitly asks the user to compare two alternatives.

- I develop a new machine learning algorithm, based on maximum-margin techniques, for estimating the user's cost function from the feedback provided. The empirical evaluation shows that my method is substantially faster than a Bayesian approach yet produces equivalent results.

- I investigate question-generation algorithms for the active elicitation interaction, proposing two solutions: one based on analyzing concrete outcomes and the other performing computation entirely in the parameter space.

- I present the results of a user study showing that ARNAULD is a useful tool for designers with variety of backgrounds.

- I argue that ARNAULD is a general tool, applicable to most optimization-based systems. I demonstrate ARNAULD's utility by integrating it with SUPPLE, but also explain how ARNAULD can be used with RIA, context-sensitive notifications, and a decision support system.

---

[1] ARNAULD is named after Antoine Arnauld (1612–1694), a French philosopher who first applied the principle of Maximum Expected Utility [7].

4.1 Design Requirements

Utility theory stems from the notion of *preferences* over *outcomes* [74]. Outcomes result from the choices of the system or the user, and the user's preferences are defined in terms of an order, $\succ$, over these outcomes. This preference order can be defined in terms of a real-valued *cost function*, $\$$, over outcomes — one prefers o over o' (written $o \succ o'$) if and only if o has lower cost: $\$(o) < \$(o')$. While people are capable of specifying preferences between concrete outcomes, they have difficulty articulating a real-valued cost function [23]. Because applications typically need an actual cost function, I seek a tool which can automatically construct a good one from a set of concrete preference examples.

Depending on the application, this cost function might denote many things. In LineDrive, cost measures the estimated cognitive difficulty of a user reading a map [3]. In the Automated Travel Assistant, ATA, cost reflects the undesirability of a sequence of airline flights [81]. In the Responsive Information Architect, RIA, cost measures the degree to which a user is unsatisfied with the system's answer to her query [152]. And in Supple, cost measures the difficulty of using different combinations of widgets for user interface tasks.

Typically, these cost functions are defined in parameterized form, in which the parameters represent various tradeoffs. Many decision-theoretic systems make strong claims to extensibility. However, while it may be easy to add new components to these systems, it is often hard to pick cost parameters which correctly integrate the new features. This creates a possible source of error that the designers of the original platform cannot control. In general, it is very hard to choose parameter values in a way that accurately orders a wide variety of outcomes. Furthermore, many of these parameters reflect *subjective* judgments, which are potentially controversial. Since these parameters need to be set by humans, certain values (or constraints on values) may be set incorrectly or inconsistently — yet the overall system must degrade gracefully in the face of such contradictions.

In summary, choosing the appropriate weights for a parameterized cost function has long been acknowledged to be challenging. I conclude that any tool that facilitates the elicitation problem must satisfy the following desiderata:

1. Make it *fast* and *easy* for both developers and end-users to find good values for the weighting parameters.

2. Output weights which are *correct* and *robust*, even in the face of erroneous or inconsistent user feedback.

I present ARNAULD in the context of personalizing SUPPLE's automatic user interface generation process. Recall, that SUPPLE's cost function was defined in Section 3.3.2 as follows:

$$\$(R(\mathcal{S}_f), \mathcal{T}) = \sum_{e \in \mathcal{S}_f} \sum_{k=1}^{K} u_k f_k(R(e), \mathcal{T}) \tag{4.1}$$

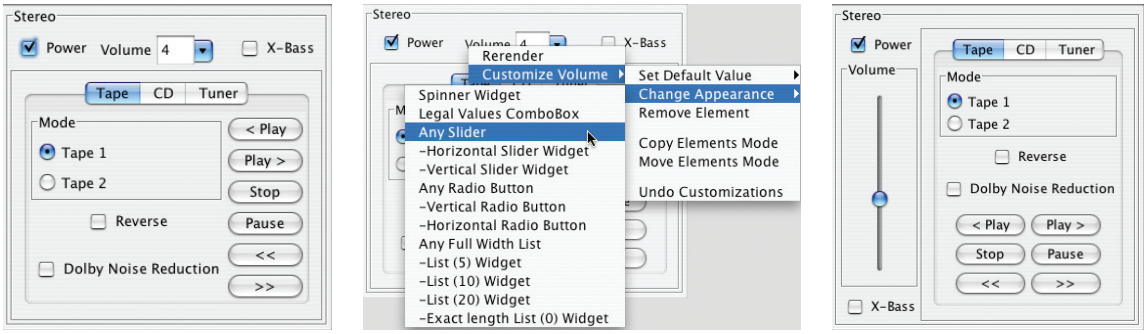
Here e denotes an element of the functional specification and $R(e)$ denotes the concrete widget that the rendering function R has chosen for e. Lastly, $f_k(R(e))$ is one of the K factors comprising the cost function and u_k is a corresponding weight. Factors can be either binary indicator functions reflecting presence or absence of a property, or they can return a non-negative[2] real value to reflect the magnitude of a property. For example, there are binary factors to indicate that a widget is a checkbox or that it is a spinner assigned to represent a numeric value. Continuous-valued factors include, for example, the downsampling factor for images and maps, or the number of lines by which a list widget is too short to accommodate the expected number of elements.

SUPPLE relies on about 50 factors, and the values of the corresponding weights represent complex interface design trade-offs. Choosing these weight values manually was a tedious and error-prone process, and it had to be partially repeated every time a new kind of widget was added to any of the platforms supported by SUPPLE.

4.2 User Interactions

In this section, I demonstrate two classes of interactions for eliciting users preferences, which we encode as inequalities of the form $o \succ o'$, indicating that user prefers *outcome* o

[2]Although SUPPLE requires factors to return non-negative values, ARNAULD imposes no such restrictions.



(a) (b) (c)

Figure 4.1: Example critiquing in SUPPLE. (a) Initial rendering of a stereo controller. (b) The user asks SUPPLE to use a slider instead of a combo box for controlling the volume. (c) The new rendering. Information recorded from this interaction is used by ARNAULD to further improve SUPPLE's cost function for future renderings.

to o'. These preferences are used, as the next section explains, to calculate a real-valued cost function, which is maximally consistent with the preferences and which can be used to generate good interfaces in the future.

The first elicitation method, called example critiquing, is well suited for both developers and end users, because such critiques may often be derived implicitly from normal user actions. The second, active elicitation, is a computer-driven questioning process where the human operator is asked to make comparisons between different pairs of automatically chosen interfaces.

4.2.1 User-Driven Example Critiquing

In many interfaces, the user's natural interactions provide information about their cost model; in these cases, one may learn the cost function without imposing any additional burden on the user using a process often called *example critiquing* [81, 116] (also referred to as *tweaking* [21, 90]). In the context of SUPPLE, its extensive customization framework (Section 3.7) allows both designers and users to modify the structure, behavior, and appearance of the rendered user interfaces. Among other things, SUPPLE's customization facility allows

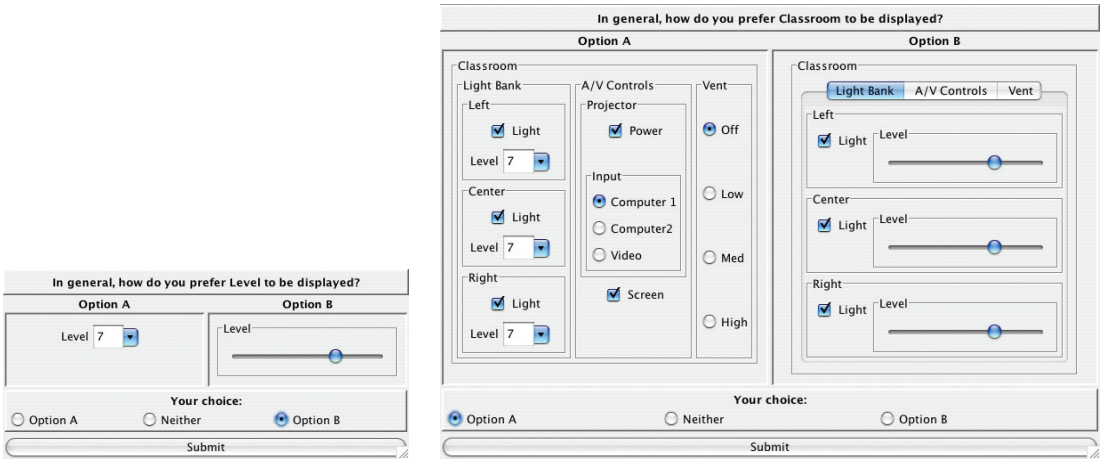
human operators to right click on any part of a interface and choose from a selection of possible ways that element can be rendered (Figure 4.1). I use ARNAULD to automatically record all such customization requests and use them to further improve the parameters of SUPPLE's cost function. Note, however, that changes to one small part of a user interface (e.g., causing light intensities to be rendered with sliders instead of combo boxes) may cascade, causing changes elsewhere (e.g., causing the top level pane of the interface to be split into separate tabs, thus making the navigation through the interface more difficult). Thus, a local improvement might lead to a global decrease in quality. In order to correctly learn the parameters of the objective function (i.e., to account for these tradeoffs), it is important to also record whether or not the user considers the resulting interface a global improvement over the previous version. For this reason, when ARNAULD detects that critiquing has caused non-local changes to the outcome, it requests feedback on the global result in addition to the local choice.

4.2.2 System-Driven Active Elicitation

In some cases, the user's natural actions provide insufficient feedback to learn a cost model. In these cases, the computer must facilitate preference elicitation by generating information-gathering questions to ask the user. Some researchers advocate rating outcomes on continuous scales or ranking multiple outcomes, but recent work showed that binary choice queries (where users are presented with pairs of outcomes and asking which of the two, if either, they prefer) result in more robust responses [23].

To keep questions simple for the user, I believe it is important to emphasize so called *ceteris paribus* (everything else being equal) queries, where users are asked to consider two small differences in isolation (Figure 4.2(a)). However, whenever the local change between the two options causes cascading global changes, ARNAULD also asks the user to compare the entire outcomes (Figure 4.2)(b).

The key challenge for active elicitation is determining the best questions to ask the user. I explain my question-generation algorithm in Section 4.4, but first I must explain how ARNAULD uses the *answers* to these questions to learn a cost function.



(a) (b)

Figure 4.2: Two consecutive steps in the active elicitation process. (a) ARNAULD poses a *ceteris paribus* query, showing two renderings of light intensity control in isolation; this user prefers to use a slider. (b) Realizing that the choice may impact other parts of the classroom controller interface, ARNAULD asks the user to consider a concrete interface that uses combo boxes for light intensities but is able to show all elements at once, and an interface where sliders are used but different parts of the interface have to be put in separate tab panes in order to meet the overall size constraints.

4.3 Learning from User Feedback

Now that I have described how to elicit a set of user preferences over pairs of outcomes, $\{o \succ o'\}$, I formally represent these preferences as constraints and use them to infer the best values of the cost function's underlying parameters. In every system I have studied, the cost (or utility) of any outcome can be decomposed linearly with respect to any class of parameters and consequently represented as

$$\$(o) = \sum_{k=1}^{K} u_k f_k(o) \tag{4.2}$$

where $f_k(o)$ is a *factor* in the cost function. Like in SUPPLE, these factors represent the presence, absence, or magnitude of various properties of the solution. In SUPPLE, one factor

is used to indicate that a particular widget is a spinner while another reflects how much an image was scaled from the desired size. In a probabilistic system like [66], these factors correspond to probabilities. The u_k is a non-negative weight corresponding to factor f_k — it is the values of these weights that must be estimated.

4.3.1 Turning Feedback Into Constraints

One can naturally convert a user's preferences over a pair of outcomes o and o' into a constraint by noting that if $o \succ o'$ then the cost of o' must be greater than that of o:

$$\sum_{k=1}^{K} u_k f_k(o') \geq \sum_{k=1}^{K} u_k f_k(o) \tag{4.3}$$

Because the factor functions always return the same values for the same outcomes, the goal is to find weight values, u_k, which satisfy the constraints.

Consider the simple example of Figure 4.2(a). In the rendering on the left hand side, all factors are set to 0 except $f_{\textit{combo box}}$ and $f_{\textit{combo box for number}}$, which are set to 1, indicating that a combo box has been used and it was used to provide access to a numeric state variable. On the right hand side, only two factors again are set to 1: $f_{\textit{slider}}$ and $f_{\textit{horizontal slider}}$. Because the user stated that she preferred sliders to combo boxes, Equation 4.3 becomes:

$$u_{\textit{combo box}} + u_{\textit{combo box for number}} \geq u_{\textit{slider}} + u_{\textit{horizontal slider}} \tag{4.4}$$

While it is clear how to formally interpret user's responses to the active elicitation queries, the example critiquing feedback deserves additional attention. Note that if the user is presented with an interface such as the one in the right pane of Figure 4.2(b) but wishes SUPPLE would render it as shown in the left pane of Figure 4.2(b), she can proceed in two possible ways: she can use SUPPLE's critiquing facility (as in Figure 4.1) to tell it not to use the tab pane to organize the top level of the interface, or she can request that the light intensities be rendered with combo boxes instead of sliders.

In the first case, her feedback reflected her overall preference for avoiding tab panes wherever possible. In the second case, she chose a less desirable widget for the light intensity in order to improve the overall interface. In order to formalize the preference as a correct

constraint, ARNAULD needs to be able to determine which of the two was intended. To perform this disambiguation, I assume that the vast majority of the feedback events will be of the first kind, where the user's preference for one widget over another will apply in any situation.[3]

In some cases (e.g., [26]) it may be reasonable to assume that if the user critiques an example by expressing a preference for one outcome over another, then the preferred outcome can be interpreted as being better than *all* other possible outcomes (rather than just the one it was explicitly stated to dominate). Because this interpretation is not always correct, ARNAULD takes the narrow interpretation.

4.3.2 Defining the learning problem

My objective is to find values for each weight, u_k, such that all constraints (or as many as possible) are satisfied. To make sure that the cost function is robust, I also wish to maximize the degree of satisfaction of the constraint, c_i. This degree of satisfaction is called the *margin* and denoted m_{C_i}:

$$m_{C_i} = \sum_{k=1}^{K} u_k f_k(o'_{C_i}) - \sum_{k=1}^{K} u_k f_k(o_{C_i}) \tag{4.5}$$

One way to find the weight values is with the Chajewska et al. approach [24], which finds the Bayesian estimate of these values given user's responses. This method relies on Metropolis sampling, which is too slow for interactive use, as I demonstrate in Section 4.5.2. Consequently, I propose a faster and more direct technique which explicitly tries to maximizes the margin through linear optimization.

4.3.3 Maximum Margin Learner

My algorithm is based on maximum-margin methods from support-vector machines [20]. The maximum-margin approach finds the factor-weight values which satisfy the most constraints. Furthermore, these values are chosen so as to maximize the difference between

[3] Another approach would be to provide a user interface mechanism for disambiguating the two interpretations or attempting to automatically predict the user's intention.

the two sides of the constraint inequality. At the same time, explicit *slack variables* are used, which allow the optimization to succeed even in the presence of contradictory or hard-to-satisfy constraints.

Formally, I reformulate the constraints from Equation 4.3 to include the addition of a shared margin variable m and a per-constraint slack variable ξ_i:

$$\sum_{k=1}^{K} u_k f_k(o'_{C_i}) - \sum_{k=1}^{K} u_k f_k(o_{C_i}) \geq m - \xi_i \quad \text{for all } c_i \tag{4.6}$$

Because I constrain the margin to be greater than or equal to 1, the sum of slack variables, $\sum \xi_i$, is an upper bound on the number of violated constraints. Thus, I formulate an objective function to be of the form:

$$\text{maximize} \quad m - \alpha \sum \xi_i \tag{4.7}$$

Where α stands for a parameter that controls the tolerance of the learner to a small number of violated constraints or constraints satisfied by a margin smaller than m.

The set of parameters returned by this optimization satisfies the maximum number of constraints and satisfies them by the maximum possible margin. Further notice, that all the constraints and the objective function are linear and therefore ARNAULD can solve this problem using very fast linear programming techniques.

Unlike a Bayesian learner [24], a max-margin algorithm can utilize prior knowledge only if it is presented in the form of constraints — it does not allow the knowledge to be summarized in the form of concise statistics. To address this shortcoming, I extend my approach so that it can take into account prior knowledge. Suppose ARNAULD has a prior belief (e.g., factory preset) that a weight for the k^{th} factor is u'_k, then it may be desirable for any new value u_k to be *close* to the prior value. This can be formalized as the following optimization problem:

$$\begin{aligned} &\text{minimize} && m' + \alpha' \sum \xi'_k \\ &\text{subject to} && |u_k - u'_k| \leq m' + \xi'_k \ \text{ for all } k \in [1 \ldots K] \end{aligned} \tag{4.8}$$

Because this constraint is not linear, I rewrite it as a pair of linear constraints:

$$\begin{aligned} u_k - u'_k &\leq m_{prior} + \xi'_k \\ u'_k - u_k &\leq m_{prior} + \xi'_k \end{aligned} \tag{4.9}$$

Finally, I combine the prior knowledge and the knowledge from the newly acquired constraints as the final objective function:

$$\text{maximize} \quad \frac{N}{N+N'}\left(m - \frac{1}{\sqrt{N}}\sum_{i=1}^{N}\xi_i\right) - \frac{N'}{N+N'}\left(m' + \frac{1}{\sqrt{K}}\sum_{k=1}^{K}\xi'_k\right) \tag{4.10}$$

I have replaced the previously undefined α and α' from Equations 4.7 and 4.8 with the inverse of the square root of the number of constraints (those collected from the user, and those represented in the prior). Additionally, I have weighted the parts of the objective function corresponding to the new constraints and those in the prior. N' stands here for the *equivalent sample size* — setting N' to 10 would mean that the knowledge included in the prior should be treated with as much weight as 10 new constraints. This allows ARNAULD to encode the certainty about the knowledge represented by the prior.

4.4 Generating Query Examples

As discussed previously, one way in which ARNAULD elicits feedback from the user is by presenting her with two alternative outcomes and asking for a statement about her preference. It should be no surprise that the choice of questions asked will affect the number of questions necessary to learn a good cost function. For example, repeatedly asking very similar questions will provide too little information for the learning algorithm to make much progress. Unfortunately, choosing the *optimal* question from the perspective of expected information gain is intractable [15]. Furthermore, complex questions are usually harder for users to answer than simple ones. Specifically, asking a user to compare two vastly different outcomes may result in an ambivalent response ("apples *vs.* oranges"), because of the vast number of tradeoffs. Thus, I seek heuristic query-generating algorithms that will ask simple and informative questions.

I assume that the user provides a set of *training examples*, each being a sample input to their application. In the case of learning the cost function for a SUPPLE device description, each training example is a functional specification of an interface and a screen size constraint. For LINEDRIVE, an example would be a route, and for RIA an example would be a set of information to be presented to the user. Note that each example may be decomposed into primitive *constituents*, corresponding to the smallest portion of the example that can be used as an argument to the cost function. With SUPPLE each element (node in the functional-specification tree) is a constituent for which a widget or a layout must be assigned and costed. In order to generate simple, localized questions, ARNAULD only issues queries about single constituents.

4.4.1 Weight-Based Question Generation

My weight-based ranking algorithm applies to any application that can enumerate the space of queries. Thus, when applied to SUPPLE, ARNAULD enumerates all pairs of different renderings of any *single* element in the interface specification of an example (i.e., varying either a single widget or the layout of one portion of the interface). These queries are ranked via a computation in the space of possible factor weights, and the best is chosen.

The key observation is as follows: if the cost function is defined in terms of K weights, then the constraints thus-far recorded define a sub-region of this space, and the region's centroid denotes the K weight values, which define the current, best cost function. Every candidate query $(o \ ? \ o')$ defines a *hyperplane* in this space—the set of weight values that cause $\$(o) = \$(o')$. Hence, an answer to the query (i.e., a statement of the form $o \succ o'$) transforms the query into a new constraint: the half-space to one side of the hyperplane is consistent with the constraint while the other is not. It has been shown that if a question is very good, then its corresponding hyperplane falls very close to the region's centroid. This is intuitive, because no matter what the user answers, the region containing the true cost function will be cut nearly in half [125, 139].

But there may be several hyperplanes, each passing close to the centroid, but oriented differently from each other, perhaps even orthogonal; which is best? Intuitively, one would

like the constrained region to be approximately spherical. Thus, if the region is already narrowly constrained along one dimension, the next cut should be orthogonal. Put another way, on average, the questions should equally constrain *every* dimension of the weight space.

My algorithm exploits these intuitions using a heuristic encoded as a modified distance function. ARNAULD ranks queries by the distance between the corresponding hyperplane and the centroid, but the distance function scales each dimension by a value proportional to the number of constraints cutting along that dimension. While this method is heuristic, it is fast to compute and effective.

4.4.2 Outcome-Based Question Generation

This algorithm applies to applications satisfying the condition that each factor of the cost function corresponds to a single constituent (hence the cost of an example may be written as the sum of the costs of the example's constituents). For instance, in SUPPLE one may create a new example by taking a constituent (element) from one functional specification and considering that element in isolation (i.e., computing the best rendering of an interface with a single widget corresponding to that element).

My outcome-based ranking algorithm first enumerates all training examples and uses the current cost function to choose the best outcome for each example as a whole. Next, for each constituent e it notes what partial outcome o_e was assigned to it in the solution that addressed the example. Then, it iterates over all possible partial outcomes o'_e that could have been assigned to e and computes $score(o'_e) = \$(o_e) - \$(o'_e)$. Note that if *score* ends up being positive, then it means that o_e is a suboptimal local outcome for e, and it was presumably chosen in order to allow for a better solution in a different part of the interface. If the *score* is negative then o_e was the better choice according to the current cost function. My algorithm chooses query $(o'_e \ ?\ o_e)$ where $score(o'_e)$ is the highest. For positive values of *score*, this challenges the tradeoff that the interface generator made. For slightly negative values of *score*, it asks about a close, second-best, local outcome, thus helping to either spot incorrect choices or widen the margin between the two options.

4.5 Evaluation

I conducted a sequence of experimental studies to measure how well ARNAULD meets the desiderata, to evaluate my learning method, and to compare my two question-generation algorithms. Recall that I presented two design requirements: 1) Make it *fast* and *easy* for both developers and end-users to find good values for the weighting parameters; and 2) Output weights that are *correct* and *robust*, even in the face of inconsistent or erroneous user feedback.

4.5.1 Informal User Evaluation

The first design requirement concerns ease of use for both developers and end-users, which I evaluated in a pilot user study. In this study, I observed four participants using ARNAULD connected to SUPPLE. Two of the four participants represented developers — they were experienced contributors to the SUPPLE project, but they were unfamiliar with ARNAULD. The other two participants represented "sophisticated users" — experienced programmers who were unfamiliar with ARNAULD and who had only cursory knowledge of SUPPLE. In Chapter 5, I discuss the results of using ARNAULD with a larger number of users in a more realistic setting.

I gave the "sophisticated user" group an overview of SUPPLE's behavior and its overall architecture followed by instruction into the operation of ARNAULD, including the ability to switch between the question generation (driven by the domain-dependent algorithm) and example critiquing modes. I then asked them to create a cost function for a new SUPPLE interface platform using ARNAULD until they were satisfied that SUPPLE was creating reasonable user interfaces.

I treated the "developer" group slightly differently by first asking them to *manually* assign values to all parameters in the cost function. (One of the participants had done this in the past but for a different set of parameters describing a different interface platform.) I then instructed them in the use of ARNAULD, and asked them to use it to construct the cost function over again, from scratch. In addition to monitoring their overall use of and satisfaction with ARNAULD, I also recorded each preference response and compared them

to those that their hand-crafted cost functions would have generated. Interestingly, the participants disagreed with their hand crafted preference model 26% and 22% of the time, respectively. Additionally, in a further 12% and 9% of the cases their hand-crafted cost function disagreed less strongly (i.e., one expressed a preference while the other considered the two options to be equivalent). This certainly shows how difficult it is to manually define a cost function. In contrast, the results of a subsequent study conducted with 17 naive users (Section 5.7.2) showed that, on average, cost functions generated automatically by ARNAULD disagreed with only 7.5% of users' preference statements.

All four participants felt that ARNAULD was a much easier and more accurate way to find a good set of cost parameters. They attributed the improvement to being able to improve the function in a rapid, iterative manner with immediately visible feedback. The "sophisticated users" felt confident that they could produce a robust set of parameters, while the developers felt that the tool let them arrive at the right result more quickly and often helped them address trade-offs, which they had missed when manually setting parameter values.

All participants were able to produce robust cost functions in 10 to 15 minutes, while the SUPPLE developers took 20 to 40 minutes to construct cost functions by hand. All participants found the tool generally easy to use, but some of them pointed out that when ARNAULD was asking them to compare more complex outcomes it was sometimes difficult to pinpoint all the differences. On the other hand, less than 10% of their answers to the automatically generated queries were ambivalent, indicating that the differences between presented alternatives were usually easy to evaluate. Two of the participants also commented that a closer blending of the implicit and explicit approaches would be desirable, that is, being able to modify the examples provided in the active elicitation or being able to more quickly alternate between the two modes. One person requested an *undo* button which would reverse a critique or answer and also remove the corresponding input from the learner.

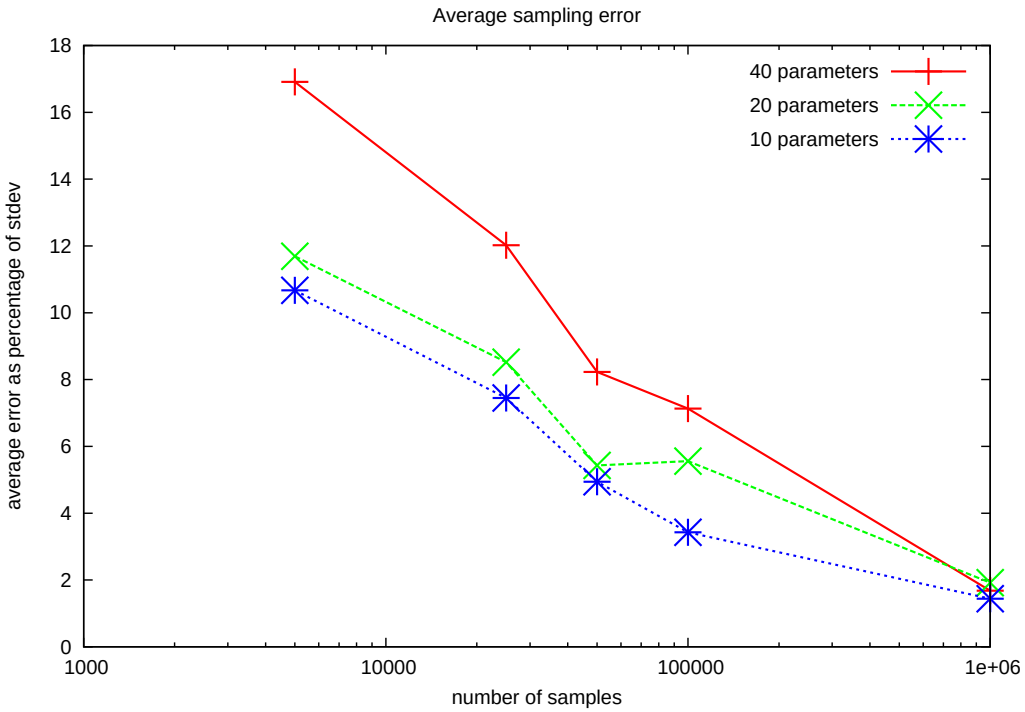


Figure 4.3: The sampling error of the Bayesian approach diminishes with a huge number of samples, but is thus too slow for interactive use: accurate answers require 10^6 samples (taking 40 seconds to compute), while my maximum-margin algorithm returns exact answers in less than 200ms.

4.5.2 Comparison of Learning Methods

The first design requirement calls for fast and easy operation, so I consider the speed and accuracy of my learning algorithm, comparing it to the Bayesian approach with Metropolis sampling, proposed by [24]. When the Metropolis sampling approximation was run long enough, both methods produced essentially the same parameter weights. But my maximum-margin approach not only produced exact answers, it was also much faster.

Because Metropolis sampling is an anytime approximation method, one may trade solution quality for speed. Perhaps it generates relatively good answers quickly? Figure 4.3 shows the average error in the Metropolis sampling estimates as a percentage of the standard deviation of the underlying distribution for the cases where 10, 20 and 40 variables need to be estimated at a time. My implementation of the Metropolis algorithm can sample 25,000 times per second. But my maximum-margin algorithm can produce an exact solution in 40-200ms on a standard desktop computer.[4] Thus, my the sampling algorithm can draw only 5,000 samples in the time used by the maximum-margin learner. When estimating 40 variables at once (which is close to the number of parameters learned for SUPPLE), this sample size produces an average error of 17% of the standard deviation of the underlying distribution. The error goes down to 12% and 8%, if one is willing to wait 1 or 2 seconds for the result, respectively. The error stays above 2% unless a million samples are drawn, which takes about 40 seconds. I conclude that maximum-margin is the clear choice for the fast interactive use required by ARNAULD.

4.5.3 Comparison of Query-Generation Methods

Another aspect of the first design requirement concerns *how many* questions need to be answered. Thus, I evaluate my query-generation algorithms to see how many interactions are required in order to find a good set of values for the 40 parameters of the SUPPLE cost function.

In these experiments, I simulate the user interacting with ARNAULD's automated querying interface (but not with example critiquing). I use a cost function defined by a previously validated set of weights (denoted the *target* function) in order to simulate a user's responses. When presented with a query, the simulator registers a preference for the option which has lower cost according to the target function. After each response, ARNAULD updates the weights of the cost function *being learned*. The quality of the learned cost function is assessed at each step by using it to generate several concrete interfaces. The target function is then used to score those interfaces. I compute

[4] Apple Dual G4, 877MHz, 1.25GB RAM

$\$(R(\textit{test interfaces}, \$'))/\$(R(\textit{test interfaces}, \$))$

where $\$$ is the target cost function and $\$'$ is the function being learned. In other words, I compute the ratio of the true cost of the rendering guided by the learned cost function, and the true cost of the rendering guided by the true cost function. This quotient reflects how far from the ideal are the interfaces generated using the currently learned parameter values.[5] To make the testing fair, I use different interface specifications to generate queries and to evaluate the results. Queries are based on the Amazon search and classroom interfaces (see Figures 3.4 and 3.5), while evaluation was based on an interactive map-based interface, an email client and a stereo controller (Figures 3.2, 3.3 and 4.6).

Figure 4.4 shows how the quality of the learned function improves with the number of queries issued. The results presented are averaged over 10 runs and include a comparison to a base line random query generator. My outcome-based and weight-based algorithms resulted in learned functions performing almost identically to the target function after about 25 queries. To better understand the properties of these query generation algorithms, I performed pairwise statistical analyses[6] of their performance separately on two intervals in the later part of the elicitation process: from 31 to 40 elicitation steps, and from 41 to 50. In the interval from 41 to 50 elicitation steps, there is a small but statistically significant between these two methods, with the weight-based algorithm producing slightly better results. There was no significant difference between these two methods in the region from 31 to 40 elicitation steps. As is quite clear from the figure, the random query generator resulted in significantly worse outcomes than either of my methods.

4.5.4 Sensitivity to Input Noise

Next, I evaluate Arnauld against my second design requirement: are the results robust even in the presence of input errors and inconsistencies? For each query generation algo-

[5]Because there exists a continuum of parameter values that may all produce the same concrete outcomes, I evaluate the learned cost function based on the cost of the results it produces (rather than the actual weight values).

[6]In order to assess the statistical significance of the differences among the three methods, I used a Wilcoxon Signed Rank test with Holm's sequential Bonferroni procedure [130]. The results are reported at the $p = .05$ significance level.

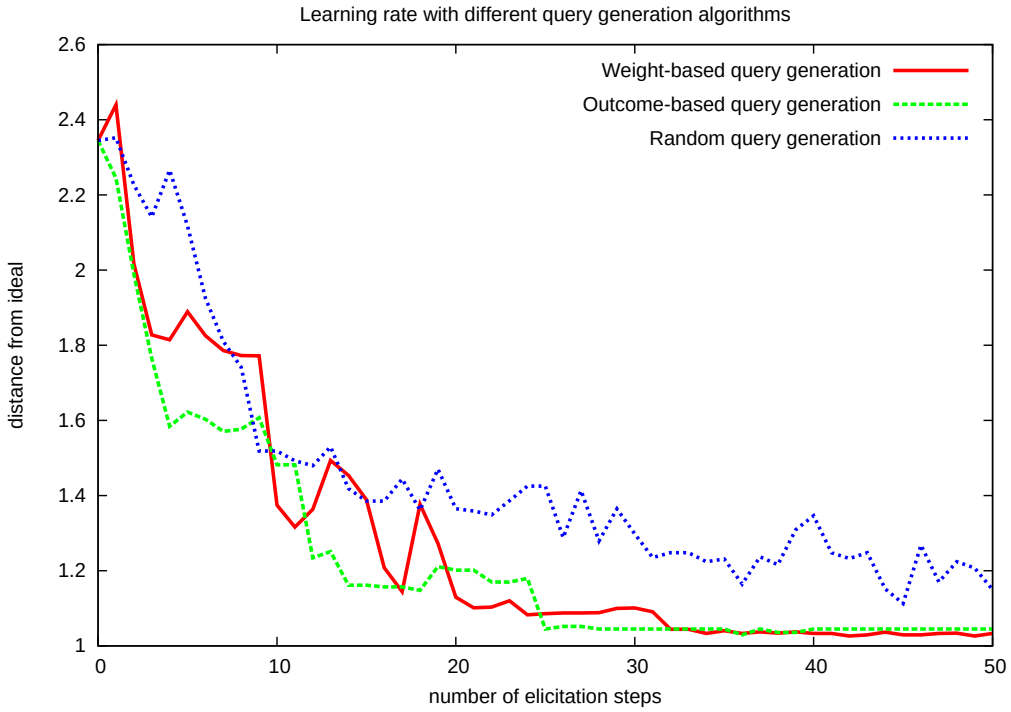


Figure 4.4: Rate of learning for different query generation strategies (all results averaged over 10 runs). The y-axis shows the cost (calculated using the *target* cost function) of the best interface generated using the learned cost function divided by the cost of the best interface generated using the target cost function. 1 = ideal.

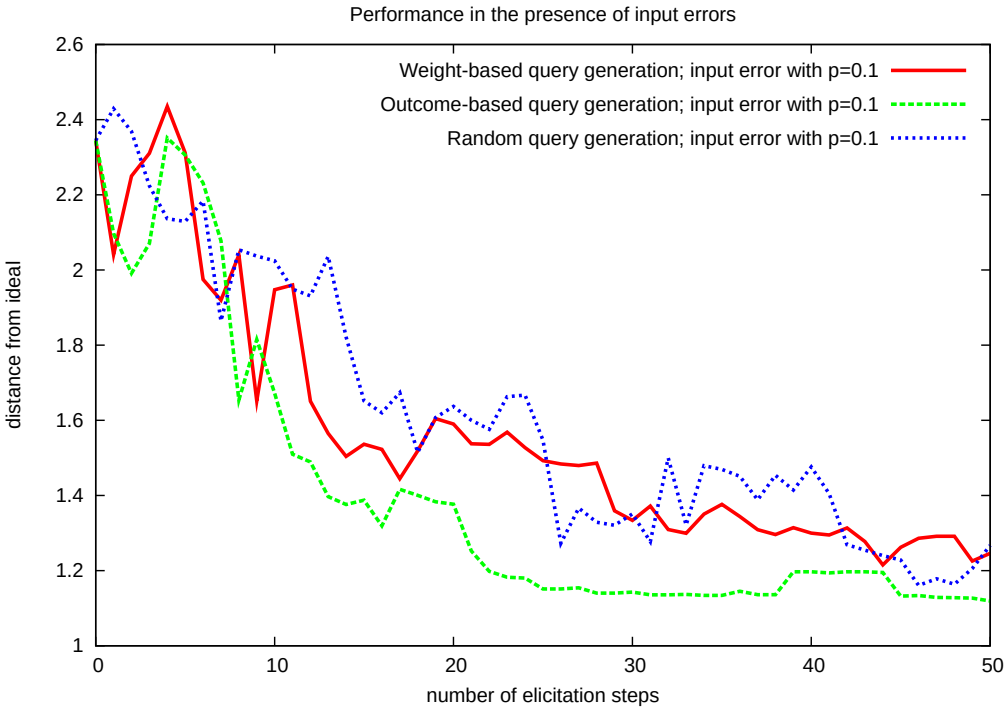


Figure 4.5: Rate of learning in the presence of input errors: at each step there was a 0.1 chance that the simulated user would give an answer opposite from the one intended (all results averaged over 10 runs). The y-axis shows the cost (calculated using the *target* cost function) of the best interface generated using the learned cost function divided by the cost of the best interface generated using the target cost function. 1 = ideal.

rithm, I repeated the experiment from the previous subsection, observing how the system behaved in the face of 5% and 10% probability of an input error at each step (i.e., a situation where the response suggested by the target function was flipped to its opposite). This simulates the user providing inconsistent input or making a mistake. Figure 4.5 shows the results for the 10% error probability.

The outcome-based algorithm still results in an interaction where nearly-optimal results are obtained after about 25 interactions. The weight-based algorithm tends to spread the queries uniformly among all parameter values meaning that after erroneous input is received, the system waits too long before asking more about the affected weights. This delay results in the algorithm performing noticeably poorer with input errors present compared to the error-free condition. As a result, in both intervals, the outcome-based query generation algorithm produced significantly better results than either weight-based or random algorithms. And while the weight-based algorithm converged faster than the random, and thus resulted in significantly better results in the 31-40 interval, in the 41-50 interval there is no longer a significant difference between the weight-based and random approaches.

I conclude that my outcome-ranked question generation is most consistent with my design requirements, and is the preferred approach, despite its slightly higher computational cost.

4.5.5 Examples

To make the results of this chapter more concrete, Figure 4.6 shows examples of user interfaces generated with two different cost functions learned using ARNAULD. The cost function used to generate user interfaces on the left hand side was created with desktop environments in mind. The second cost function — used to generate examples on the right hand side — reflects the design concerns relevant for making user interfaces usable on with a touch panel interface, where all interactors have to be large enough to be operated with a finger.

As these examples and those in Chapter 5 illustrate, SUPPLE's cost function is expressive enough to capture distinct interface design styles and the ARNAULD system can learn those styles from a user's preference statements expressed over concrete user interface examples.

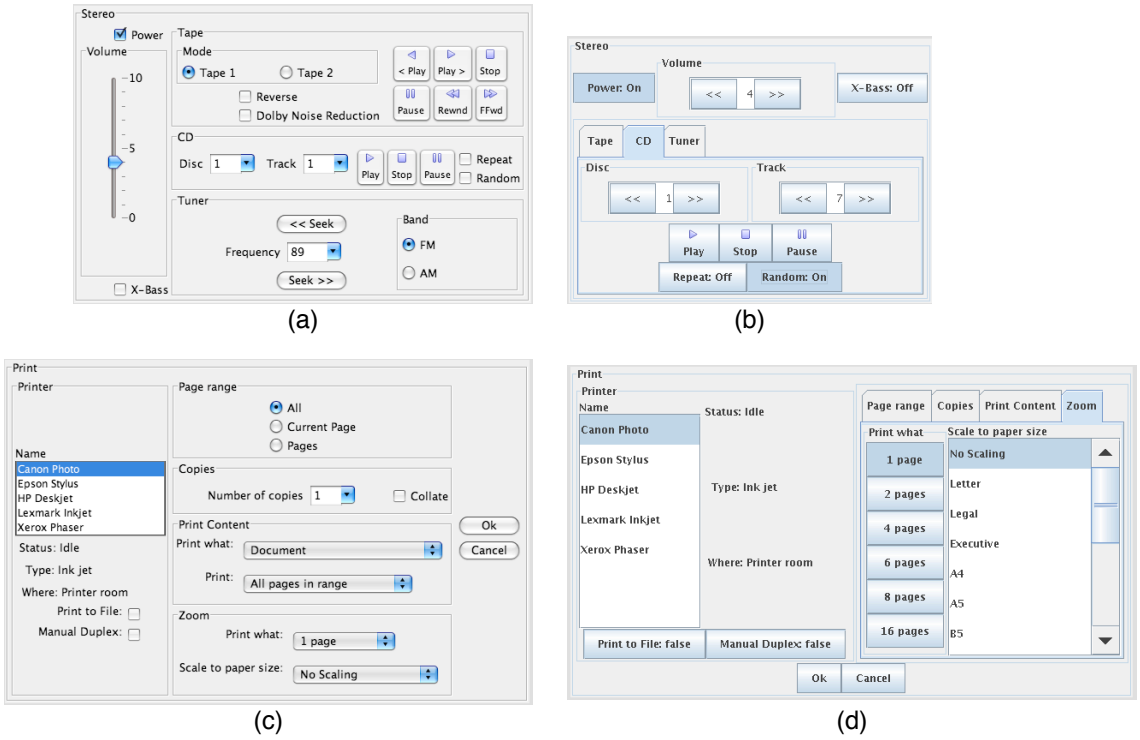


Figure 4.6: Examples of user interfaces generated automatically by SUPPLE using two cost functions elicited by ARNAULD: (left) for use on a desktop computer, and (right) for a touch panel.

4.6 Arnauld as a General Platform

Although I have only experimented with ARNAULD in the context of SUPPLE, ARNAULD will benefit most optimization-based interface systems. In this section I consider three other systems and show how ARNAULD will facilitate their operation.

4.6.1 RIA, The Responsive Information Architect

RIA is a system that automatically decides first, what information to present to the user [152], and second, how to best match different pieces of information to different modalities [153]. In the first phase, RIA balances certain tradeoffs through the use of numerous, manually-tuned parameters reflecting different classes of information used to inform the final decision.

RIA's objective function is particularly complex with many of the different parameters being multiplied by each other. Given ARNAULD's current restriction to linear systems, it could not be applied to estimate all classes of parameters at once. However, if all but one class of parameters are held constant, then the objective function would become a linear combination of the remaining class of parameters making it amenable to learning with my approach.

Both interaction methods would be appropriate in this case though a special interaction technique would have to be provided to allow users to critique the information that was delivered through speech. Also, for automated queries, speech could be simulated with text for faster interaction.

4.6.2 Context-Sensitive Notifications

BUSYBODY [66] is designed to deliver notifications to a user in a manner that is sensitive to the user's interruptability levels. It uses a decision-theoretic framework to estimate the utility of delivering a particular piece of information via the available modalities, conditioned on its estimate of the user's current interruptability. A primary focus of the work is innovative machine-learning algorithms for estimating the probability of the user being engaged in different classes of activities [67]. In contrast, the actual cost parameters, indicating the relative level of distraction incurred by different modes of notification delivery, are obtained by asking users abstractly to assign different dollar amounts reflecting how much they would be willing to pay to avoid such a notification assuming a certain level of interruptability.

The cost function driving this system relies on two classes of parameters: the costs of interrupting the user with different modalities, and the cost of delaying the delivery of a message in hopes that the user will be more interruptable later. These two classes of parameters are combined in a linear manner and thus can be learned together by ARNAULD.

Currently, this system learns its sensor model by first recording the sensory inputs for a particular user and his environment, while also recording a video of everything that transpired during that time. In order to train the system, the user has to watch the video and

annotate it with transitions between different attention states or levels of interruptability. During the same interaction, ARNAULD could ask the user queries of the form: "If I were to deliver you an important message at this point, would you have preferred if I had used email or displayed a message on the screen?" or: "If I had a message to deliver for you, would you have preferred if I delivered it immediately by displaying it on the screen or waited till later and called you on the phone?".

The resulting system could be further tuned (personalized) after deployment, if the user was given the option to comment on whatever notifications were delivered. Example critiquing could easily be supported for messages displayed on the screen but more specialized interfaces would be needed to comment on messages delivered via email or a real time phone call.

4.6.3 Parameter Elicitation For Decision Support

ARNAULD could also be used in the context of a decision support system such as Apt Decision [131], which helps users find rental apartments. Because renters' needs and preferences vary, systems like this one need to learn a new set of preferences for each end user. Currently, this systems allows users to select up to 6 positive and 6 negative criteria, which can be further ordered according to preference. The system can also generate binary comparison queries, and in response to each such query it can suggest a few parameters that might be important to the user. The system currently has no way of combining results of different comparisons or combining comparison results with users' earlier selections of important attributes. ARNAULD would provide a principled way of gathering the evidence and of assigning numeric weights to different attributes instead of just simply ordering them. Accurate weights would lead to better suggestions.

4.7 Summary

Decision-theoretic optimization has been adopted by a number of researchers to automatically generate various aspects of user interaction. Yet in almost all cases, the numerous and unintuitive parameters of the optimization's objective function are chosen by hand, thus adding an unprincipled element to an otherwise principled approach.

In this chapter, I addressed this problem by presenting and evaluating the ARNAULD system for interactively eliciting user preferences for the purpose of automatically learning parameters of optimization-based systems. ARNAULD lets users specify their preferences by providing simple feedback on examples of concrete user interfaces either through user-driven customization or through system-driven active elicitation interactions. ARNAULD uses a novel max-margin learning algorithm, which turns user feedback into a set of weights. The active elicitation interaction is driven by two heuristic query generation algorithms.

I have shown how ARNAULD allows SUPPLE to be adapted to a person's subjective preferences and illustrated this with examples of user interfaces generated with two different cost functions learned using ARNAULD. In the next chapter, I will show how SUPPLE can be adapted to a person's objective motor abilities.

Chapter 5

ADAPTING TO MOTOR AND VISION ABILITIES

Computer use is a continually increasing part of our lives at work, in education, and when accessing entertainment and information. Thus the ability to use computers efficiently is essential for equitable participation in an information society. But users with motor impairments often find it difficult or impossible to use today's common software applications [10]. Some may contend that the needs of these users are adequately addressed by specialized assistive technologies, but these technologies, while often helpful, have two major shortcomings. First, they are often abandoned because of their cost, complexity, limited availability, and need for configuration and ongoing maintenance [29, 31, 75, 113]. In fact, some prior work shows that only about 60% of the users who indicate a need for assistive technologies actually use them [31]. Second, assistive technologies are designed on the assumption that the user interface, which was designed for the "average user," is immutable, and thus users with motor impairments must adapt *themselves* to these interfaces by using specialized devices [73, 135].

Other approaches like universal design [84], inclusive design [72], and design for all [134] attempt to create technologies that have properties suitable to as many people as possible. These approaches have a laudable goal, but are often impractical, particularly where complex software systems are involved [10]. A "one size fits all" approach often cannot accommodate the broad range of abilities and skills in vast and varied user populations.

A preferable solution would be to adapt user interfaces to the measured abilities of individual users; in other words, to have the system, not the user, perform the adaptation.

Many such interfaces will be needed because of the myriad distinct individuals, each with his or her own diverse abilities and needs [10]. Note that even people with the same medical diagnosis can have a wide range of functional motor skills [69, 78]. Therefore, traditional, manual user interface design and engineering will not scale to such a broad range of potential

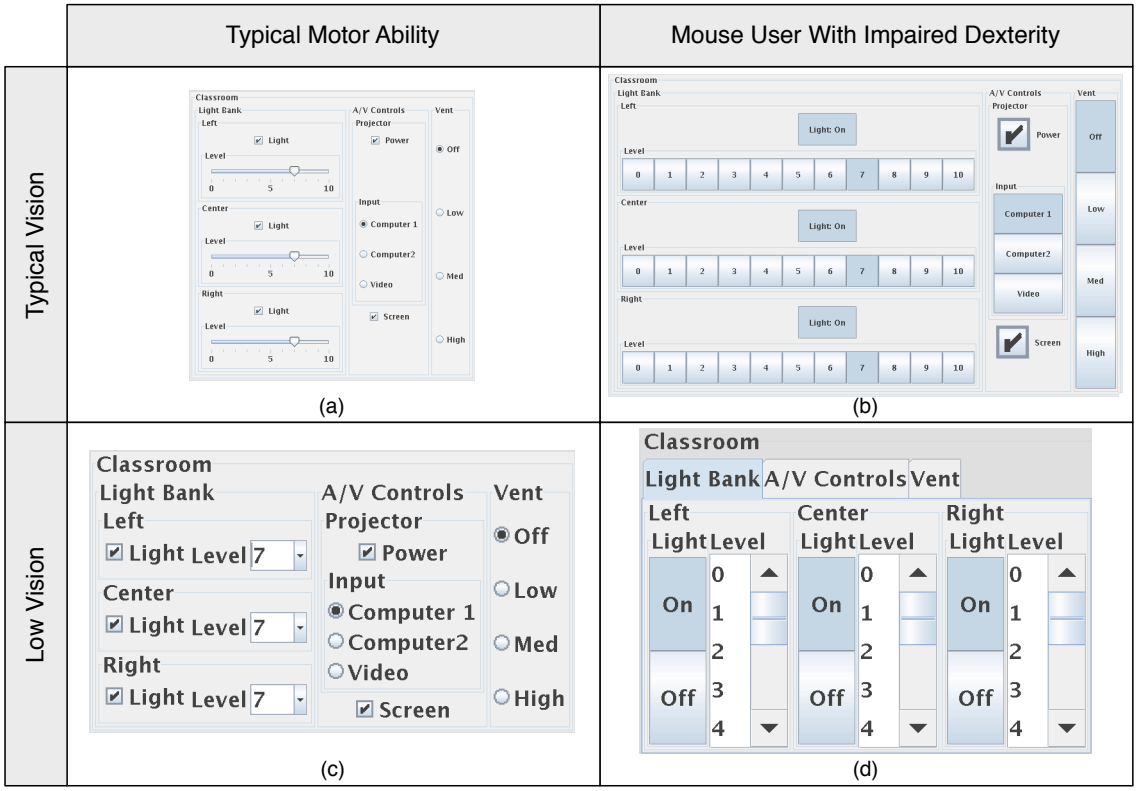


Figure 5.1: Four GUIs automatically generated by SUPPLE under the same size constraints for four different users: (a) a typical mouse user, (b) a mouse user with impaired dexterity, (c) a low vision user and (d) a user with a combination of low vision and impaired dexterity.

users. Instead, I enable GUIs to *design themselves* for users based on users' own functional abilities. This approach embraces a current trend in designing for what users can *do*, rather than for classes of users based on health conditions or presumed skill sets [27, 78].

In the previous chapter, I described ARNAULD, which can personalize SUPPLE's user interface generation process to a person's *subjective* notions of what user interfaces she prefers. This chapter introduces the ABILITY MODELER, which after directing the user to perform a one-time set of diagnostic tasks, builds a personalized model of the user's *objective* motor abilities as observed through the lens of the mouse pointer control. This personalized

model of a person's abilities is then used as input to SUPPLE to automatically generate user interfaces that are predicted to be the fastest to use for that particular person.

This chapter also introduces an extension to the SUPPLE system that allows user interfaces to be easily adjusted to accommodate users' different vision abilities. Importantly, these two types of adaptation—to motor and vision abilities—can be used together, allowing SUPPLE to adapt to people with a combination of motor and vision impairments, a population that is poorly served by current assistive technologies. As an example, Figure 5.1 shows an interface rendered by SUPPLE for four different users.

In my evaluation, I focus on the adaptation to users' motor abilities. My summative study evaluates both ARNAULD and the ABILITY MODELER by comparing user interfaces generated with these two approaches to manufacturers' default interfaces. The results show that participants with motor impairments were significantly faster, made many fewer errors, and strongly preferred the automatically-generated personalized interfaces, particularly those generated with the objective ability models, over the manufacturers' baselines.

5.1 Design Requirements

As noted, people with special needs differ widely in their motor and vision abilities [10, 69, 78]. My goal is to build a system that provides graphical user interfaces personalized to users' individual needs without any intervention on the part of designers or assistive technology specialists. Also, the system must be simple and fast to setup, use, configure, and maintain.

Currently, I focus on personalization for users who have difficulty controlling the mouse pointer (due to motor impairment, context, or input device) and users with low visual acuity. At this stage, I do not address text entry, blindness, or cognitive impairments.

A user's ability to control a GUI with a mouse depends on many factors. Among them are the distances among different on-screen elements, the complexity of navigation (e.g., using tabs or multiple widows), the types of operations required to use the elements (e.g., pointing, dragging, double-clicks), and the sizes of the elements. Thus, to support users with motor impairments, my goal is to provide them with user interfaces that strike the best balance among these competing factors. Complicating this are the complex tradeoffs

involved. For example, widgets that are larger may be easier to manipulate but may result in a larger interface, which requires greater travel distances. SUPPLE's optimization approach is a natural fit for this problem.

In order to generate user interfaces best suited for a person's individual motor abilities, SUPPLE needs an accurate predictive model of these abilities for each user of the system. A common approach is to use Fitts' law [34, 86] to model a person's mouse pointer movement time. It is debatable, however, whether Fitts' law applies to individuals with motor impairments. Prior work suggests that in some cases it might [83, 133], while in others it might not [53], and yet other studies report mixed results [18, 17, 118]. Due to these concerns, I developed a new approach for modeling people's individual movement times.

For people with vision impairments, size of the display, lighting, and distance from the screen have a large impact on usability. In such situations, the size of the visual cues can affect both task time and accuracy [127]. Most current solutions for low-vision users involve indiscriminately enlarging all the contents of the screen, for example, with a screen magnifier. Thus, useful content and empty space are enlarged equally wasting precious screen real estate. Solutions like screen magnifiers show only a fraction of the screen at a time and force serial rather than parallel exploration of the interface [77]. My goal is to generate user interfaces that are legible and that can rearrange their content so that the entire interface fits on the user's screen for efficient exploration and interaction.

A number of modern web browsers offer a "resize and reflow" feature where the user can enlarge the text on a page and the page is then instantaneously re-rendered. My goals in this chapter is to emulate this interaction for desktop GUIs in general, which requires that SUPPLE be able to redraw interfaces for different sizes interactively, and that successive renderings resemble one another as much as possible so as to reduce potential for confusion.

Finally, SUPPLE needs to support people with *combinations* of motor and vision impairments — a population that is poorly served by the current technologies because screen magnifiers, the primary assistive technology for low vision users, often make strong assumptions about users' ability to control the mouse pointer.

5.2 Modeling Users' Motor Abilities

In this section, I develop the Ability Modeler—a tool for eliciting and modeling a person's abilities to perform typical mouse pointer-driven tasks such as clicking on a distant target (pointing), dragging, performing multiple clicks, and selecting items from a list. I first present a study in which I collected motor performance data from 8 participants who exhibited a range of motor abilities and who used a variety of input devices. I then describe an automated method used by the Ability Modeler for developing and training a *personalized* predictive model for each participant.

5.2.1 Method

Participants

Eight volunteers (5 male, 3 female) aged 25 through 35 participated in the study. Six of the participants used their own personal input devices, and 3 used computers of their choosing. Table 5.1 lists the devices used and any health conditions that might affect participants' motor performance. The "code" column refers to a shorthand designation that I use throughout the rest of the chapter to refer to these individuals.

Table 5.1: List of participants. For more information about Vocal Joystick see [54].

Code	Device used	Health condition
ET01	ERICA Eye tracker (click by dwelling)	
HM01	Head Mouse (click with right fist using a switch)	spinal cord injury (incomplete tetraplegia)
M03	Mouse	muscular dystrophy (impaired dexterity)
M04	Mouse	
TB01	Trackball (Kensington Expert Mouse)	spinal cord injury (incomplete tetraplegia)
TP01	Trackpad (Apple MacBoook Pro)	
VJ01	Vocal Joystick	
VJ02	Vocal Joystick	

Throughout the rest of the chapter, I sometimes refer to *typical* users. These are users who utilize "typical devices" to interact with the computer and whose motor abilities are unimpaired (i.e., M04 and TP01). In contrast, *atypical* users will refer to those whose motor abilities are impaired (i.e., HM01, M03, or TB01) and/or who use unusual input devices (i.e., ET01, HM01, VJ01 or VJ02).

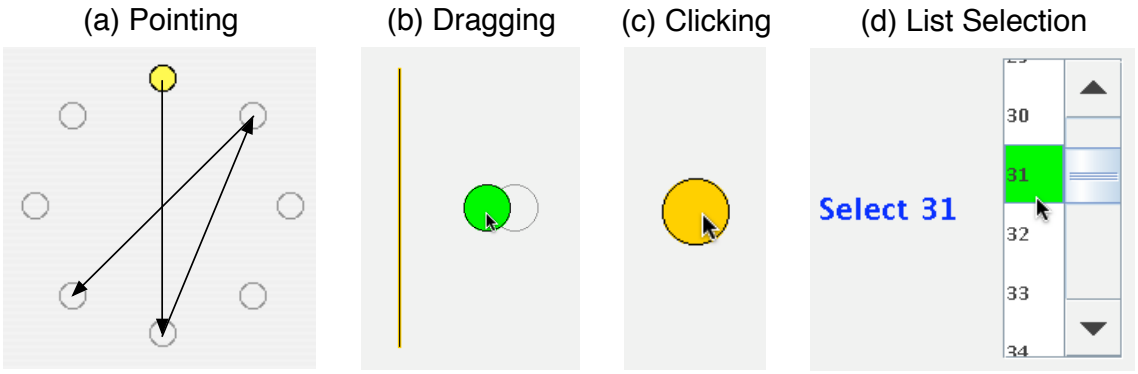


Figure 5.2: The setup for the performance elicitation study: (a) for pointing tasks; (b) for dragging tasks—here the green dot was constrained to move in only one dimension, simulating the constrained one-dimensional behavior of such draggable widget elements like scroll bar elevators of sliders; (c) for multiple clicks on the same target; (d) for list selection.

Apparatus

For measuring pointing performance, I used a set of pointing tasks based on the ISO 9241-9 standard [70] where I varied target size (10-90 pixels), distance (25-675 pixels), and movement angle (16 distinct, uniformly spaced angles). A typical task sequence is illustrated in Figure 5.2a. In all, between 387 and 572 pointing actions were elicited per participant. (Some participants who tired faster had fewer trials than other participants.) I additionally asked each participant to perform a small number of dragging and clicking tasks (Figures 5.2b and 5.2c, respectively). The study took between 20 and 40 minutes per participant.

Measures

For each action, I recorded the time taken to successfully acquire the target and the number of errors. An error would be recorded if the mouse button was pressed or released outside the target. Data from all trials (including those with errors) were used in the analysis.

5.2.2 Results and the Inadequacy of Fitts' Law

I computed Fitts' law parameters for each of the participants. The average R^2 of the resulting models for the atypical participants was .51, ranging from .14 (ET01) to .81 (VJ02). As an example, Figure 5.3a shows how movement time varied with distance and target size for participant ET01. The high variance for target sizes of 10 and 15 pixels was due to the difficulty of acquiring small targets with this particular eye tracker. Therefore, ET01's performance markedly improved at the 25 pixel target size but did not change substantially beyond 40 pixels. The distance to the target only marginally affected this participant's performance. As illustrated in Figure 5.3b, Fitts' law poorly models the observed performance of ET01. This is not entirely surprising, because Fitts' law assumes rapid, aimed movements unencumbered by issues like eye-tracking jitter.

In fact, I observed a similar lack of fit to Fitts' law for the Head Mouse user HM01. His performance degraded sharply for distances larger than 650 pixels when he could no longer perform the movement with a single head turn and had to "clutch". On the flip side, TB01, a trackball user with incomplete tetraplegia, showed the opposite trend with respect to target size. His performance improved very slowly for small targets, but these improvements became much more pronounced for sizes larger than 25 pixels. Again, Fitts' law is a poor fit for users like these.

Another matter that would complicate the use of Fitts' law occurs because SUPPLE has the option of changing the sizes of widgets in a GUI and the distances between them. These distances (D) will change as a linear function of widget sizes (W) with $D = aW + b$, where the constant term b is due to unchanged components of the widgets (e.g., labels). In this case, Fitts' index of difficulty $ID = \log_2(\frac{aW+b}{W} + 1)$ makes it clear that as W shrinks, ID grows to infinity ($\lim_{W \to 0} \log_2(\frac{aW+b}{W} + 1) = \infty$). But as W grows to infinity, ID shrinks asymptotically to a constant ($\lim_{W \to \infty} \log_2(\frac{aW+b}{W} + 1) = a + 1$). Thus, if I were to model all users using Fitts' law, a single improvement strategy would be implied for all: grow target sizes to infinity unless size constraints force dramatic adverse changes in the choice of widgets or organization. But as the diversity in my observed results imply, the participants benefit from more individualized adaptations.

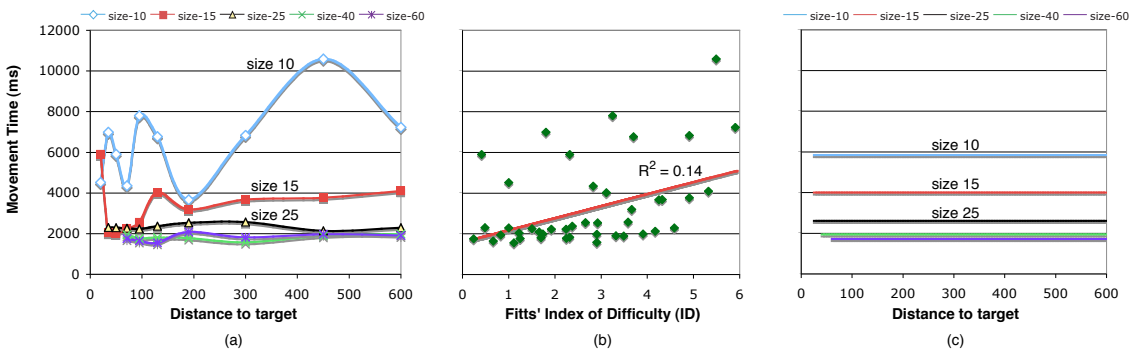


Figure 5.3: Movement time for the eye tracker user: (a) actual data, (b) Fitts' law model, (c) automatically selected custom model, which correctly captures the fact that this user was affected by the increasing target size but not by the movement distance. The same y-axis scale is used on all three graphs.

For these empirical and theoretical reasons, I proceed to develop a different method for modeling each participant's unique motor performance abilities, as described in the next section.

5.2.3 Automatically Creating Models of Pointing Performance

The Ability Modeler creates personalized models of pointing performance in two steps. First, it finds the best set of features to include in the model. In the second step, it trains a regression model that is linear with respect to the selected features.

I consider seven possible features in my model: a constant term, Fitts' index of difficulty (ID), its two individual components $\log_2(D)$ and $\log_2(W)$, as well as the raw measures W, 1/W and D. For each user's data set, the Ability Modeler evaluated the $2^7 - 1 = 127$ possible models with at least one feature using 10-fold cross validation[1] and ordered them by the mean squared error (MSE) they produced.

Cross validation is known to overestimate an algorithm's error on test data so a common practice is to look for the most parsimonious solution (i.e., one using the fewest features) in a

[1]Because data were divided randomly into the 10 folds in each instance, I repeated the cross validation process 10 times and averaged the results.

small vicinity of the optimum predicted by the cross validation [58]. The Ability Modeler thus picks the most parsimonious model whose performance is within 1% of the best model. This one-time feature selection process takes less than two minutes on a standard computer. Table 5.2 summarizes the feature results for the participants in this study.

Table 5.2: Results of the feature selection process for different participants. Although $log(W)$ was not used in modeling the performance of any of the participants, it was used in modeling some devices tested by the authors.

	D	1/W	W	log(D)	log(W)	ID	1
ET01		x	x				
HM01		x	x			x	
M03		x				x	x
M04	x			x		x	x
TB01	x		x	x			
TP01		x	x			x	
VJ01	x			x		x	x
VJ02	x	x		x			

Whereas the mean R^2 for Fitts' law models for atypical participants was .51 (ranging from .14 to .81), it improved to .71 (ranging from .49 to .88) with the personalized models. Unsurprisingly, for typical users the improvement was much smaller (from .79 to .85 on average).

Modeling Dragging and Multiple Clicks

I also explicitly model participants' ability to perform dragging operations. These are constrained to one dimension (Figure 5.2b), as is the case for many user interface widgets such as scroll bars or sliders . I also model users' ability to execute multiple clicks over targets of varying sizes.

I use the user's predicted pointing time MT_{point} (for the same distance and target size) as the only non-constant feature of the model of their dragging performance. This is because I observed that movement times for both types of operations follow similar patterns and the formulation I adopt for modeling MT_{drag} allows Ability Modeler to elicit only a

small number of dragging operations (around 40) from the user to accurately estimate the parameters of the following equation:

$$MT_{drag}(\mathrm{D}, \mathrm{W}) = a + b * MT_{point}(\mathrm{D}, \mathrm{W}) \tag{5.1}$$

The users have the option of skipping the dragging task if they cannot perform the dragging operation (the case with ET01), in which case Ability Modeler sets the a parameter to a very large number to reflect the fact that dragging was not a practical option for the user.

Equation 5.2 below shows the model I found to accurately estimate the time per click when the user has to perform multiple clicks on a target of width W.

$$MT_{click}(\mathrm{W}) = a + b * \log(W) \tag{5.2}$$

Modeling List Selection

Initially, I modeled user's operation of all complex widgets in a user interface as combinations of pointing, dragging and clicking. My subsequent pilot study (Section 5.5) indicated, however, that list selection times are poorly modeled by combinations of those primitive actions; that approach also did not allow for including the scroll wheel in the model. As a result, I extended Ability Modeler's task set and the model to explicitly include list selections. Specifically, to elicit participants' list selection performance model, I asked participants to alternately select two numbers in the list of consecutive numbers (Figure 5.2d). Both numbers to be selected were placed sufficiently far from the end of the range so that they could not be accessed when the scroll bar was moved all the way to the top or bottom. I varied the heights of the scroll window, the distance in the number of items between items to be selected, and the minimum size of any clickable element, such as list cells, scroll buttons, scroll bar elevator, or scroll bar width.

Past research [61] has demonstrated that for able-bodied users, a variant of Fitts' law holds for scrolling tasks. In that approach, the number of lines separating the start point from the target corresponds to the amplitude or the distance, while the number of lines

simultaneously visible in the window corresponds to the target width. Thus the scrolling time in lists should be directly proportional to

$$log(\frac{\text{distance measured in number of items}}{\text{size of the list widget measured in items}})$$

for typical users. Thus, taking this formulation of the index of difficulty as a starting point, I derive a set of basic features in a manner analogous to that used to derive the features for modeling pointing performance. I also use the same method for automatically selecting the most appropriate set of features for a personalized model of list selection performance.

I used the data collected from the ability elicitation part of the summative user study described in Section 5.7.1 to compare the two approaches to modeling list selection performance (the one that models list selection in terms of a combination of pointing, dragging and clicking, and the direct modeling approach). For the component-based approach, the mean R^2 for both groups of participants was only .09 (ranging from .00 to .36). In contrast, the direct model had a mean R^2 fit of .61 (range: .39-.84) for motor-impaired and .67 (.49-.76) for able-bodied participants. In light of these results, I decided to incorporate the explicit model into the Ability Modeler.

5.3 Optimizing GUIs for Users' Motor Capabilities

Given a personalized model of a user's motor performance generated by the Ability Modeler, Supple generates a GUI that minimizes the user's expected movement time (EMT) given a description of a typical task performed on that interface.

5.4 Adapting to Users with Low Vision

To adapt to a person's vision capabilities, Supple needs to vary only one parameter, namely the visual cue size. I extended Supple to allow users to directly control this parameter via a keyboard shortcut and a simple GUI, akin to the mechanism provided by most modern web browsers. This poses two challenges. First, the interaction should be nearly instantaneous. Second, as the font size varies, the changes between successive renderings must be kept to a minimum to avoid disorientation.

To address the first challenge, I added a caching system to SUPPLE and noted that only 8 discrete visual cue settings adequately cover the range from the font size 12, the default font size used in the Java Swing Metal Look and Feel, and 50. Thus, GUIs for the remaining 7 cue sizes can be pre-computed in the background while the user is inspecting the first one.

The challenge of maintaining consistency between successive renderings of an interface is addressed by the fact that SUPPLE's cost function includes an optional extension that includes a penalty for increased difference in visual appearance of a user interface from previously generated variants (Section 3.5).

Figure 5.4 shows an email client configuration GUI which has been automatically generated for a typical user (top) and for a low vision user (bottom), with font sizes enlarged by a factor of 3 and other visual cues adjusted accordingly. The GUI with large cues uses the entire area of a 1440×900 screen. While not all elements could be shown side by side and some were thus put into tab panes, the logical structure of the entire interface, as well as the action buttons at the bottom of the interface, are all visible at a glance. If instead a full screen magnifying glass were used (or if the display resolution was lowered) to achieve the same font size, only a fraction of the original interface, outlined by the dashed line, would have been visible, forcing the user to explore the contents of the GUI serially rather than in parallel.

5.4.1 Adapting to Both Vision and Motor Capabilities

I have now presented technical solutions for adapting GUIs to a user's motor and, separately, their vision abilities. These two approaches are orthogonal and combine easily to provide custom GUIs for users who are atypical with respect to *both* their motor and vision abilities (refer to Figure 5.1 for an example). The only limitation is that ability-optimized GUIs can take up to several minutes to generate so the instantaneous preview of interfaces generated for different cue sizes may not always be available, SUPPLE therefore allows users to set a system-wide preference for visual cue size and that's the first value for which SUPPLE generates new interfaces.

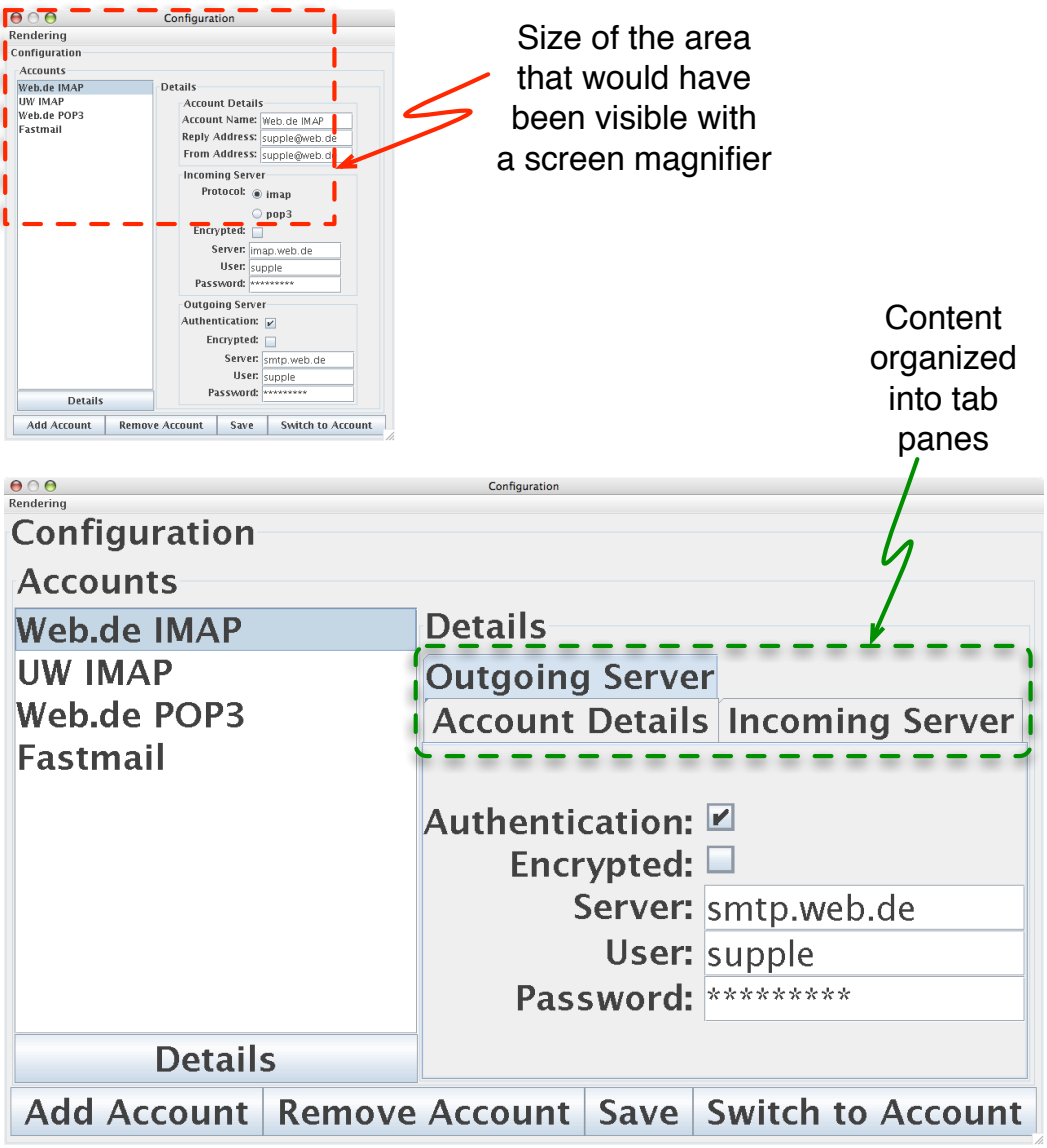


Figure 5.4: An email client configuration GUI automatically generated for a typical user (left) and for a low vision user (right) – both GUIs shown to scale. The latter interface allows the user to see the structure of the entire interface at a single glance. If a screen magnifier was used to enlarge the original interface to achieve the same font size, only a small fraction (marked with a dashed line) would have fit on the screen at a time.

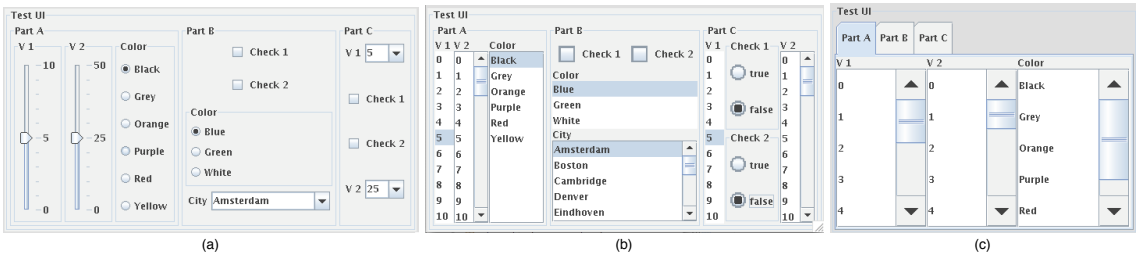


Figure 5.5: Three renderings of a synthetic interface used in the preliminary study and automatically generated under the same size constraint: (a) base line preference-optimized GUI; (b) personalized for the mouse user with muscular dystrophy (M03); (c) personalized for the eye tracker user (ET01).

5.5 Pilot User Study

I conducted a preliminary study of SUPPLE's ability to adapt to a user's motor abilities. Five of the participants who took part in the model eliciting study described in Section 5.2 also participated in this preliminary study. I used a special synthetic user interface designed so as to include many of the types of data commonly manipulated in standard GUIs. The functional specification included 11 leaf nodes: 4 numerical inputs with different ranges and input accuracy requirements, 3 choice elements, and 4 booleans. Figure 5.5 shows sample renderings. Note that in this pilot study, I used a version of the ABILITY MODELER that did *not* include the specialized model for list selection.

Each participant was presented with two sets of automatically generated GUIs. Typically, one GUI in each set was a baseline generated using SUPPLE's default cost function elicited from a small number of users experienced in interface design, one was optimized for that user's abilities (the "personalized" GUI), and one or two others were personalized for other participants for comparison. All GUIs in a set were rendered under the same size constraint (the "condition" column in Table 5.3) and the sizes were chosen to accentuate differences among renderings within each set. The participants were not told how the interfaces were generated until after the study. With each GUI, participants performed 4 brief task sets, each requiring between 10 and 12 tasks. A single task was changing a value of a

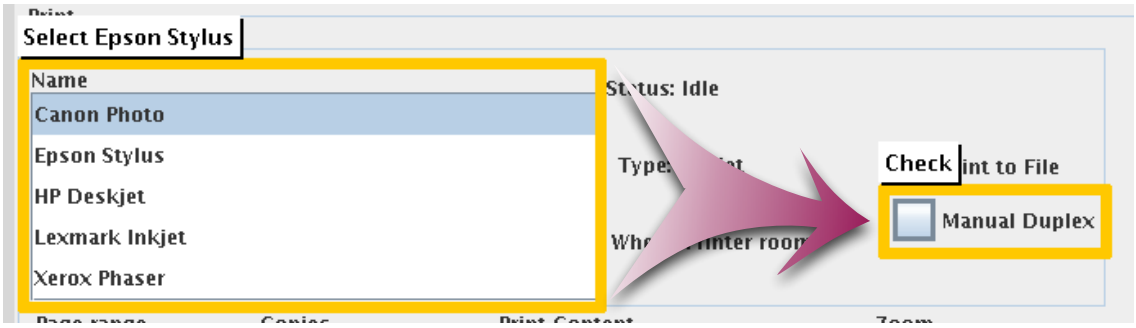


Figure 5.6: Participants were visually guided to the next element in the interface to be manipulated. The orange border animated smoothly to the next element as soon as the previous task was completed.

single widget. The first task set was a practice set and was not included in the results. I was interested in expert performance, so the participants were led through the tasks by an animated visual guide (shown in Figure 5.6), which limited the effect of visual search time (for ET01, the experimenter read the instructions aloud for each operation). Participants performed the same tasks on all interfaces and the order of interfaces (personalized, baseline, others) was randomized. Participants were instructed not to use the keyboard during the study. Some of the concrete GUIs generated for the participants are shown in Figure 5.5 and Table 5.3 summarizes the results.

5.5.1 Results

On average, the personalized interfaces allowed participants to complete tasks in 20% less time than the baseline interface, with the time differences ranging from a slowdown of 3% to a speedup of 43%. In addition, ET01 was not even able to use the baseline interface because it required dragging, which her particular software configuration did not support. In 5 out of 10 conditions, participants were fastest using a personalized GUI, and in 6 out of 10 cases, they rated the personalized GUI as easiest to use.

In 4 cases, however, interfaces optimized for a *different* participant resulted in fastest performance pointing to limitations of the early performance model. This version of the

Table 5.3: Study results (bold = fastest; underline = rated as easiest to use; VJ02 did not express a clear preference in one condition). SUPPLE allowed ET01 to complete tasks she was not able to accomplish at all with the baseline interface while for the remaining users it resulted in an average time savings of 20%.

		Average time (in seconds) taken to complete a task set			
participant	**condition**	**personalized interface**	**baseline interface**	**interfaces personalized for other participants**	
M03	small A	28.47	34.73	31.50 (ET01)	27.45 (VJ02)
	full screen	26.02	30.04	26.88 (VJ02)	
ET01	small A	56.16	not usable	48.03 (M03)	
	small B	59.04	not usable	47.95 (M03)	57.99 (VJ02)
HM01	medium A	29.96	52.40		
	medium B	29.94	47.96		
TB01	small B	43.59	42.52	47.19 (M03)	43.44 (ET01)
	full screen	47.99	48.77	39.45 (VJ02)	38.15 (ET01)
VJ02	small B	55.55	63.41	56.24 (M03)	61.52 (ET01)
	full screen	52.76	72.94	57.93 (M03)	70.45 (TB01)

model significantly underestimates the time necessary to manipulate those list widgets where only a small fraction of items is visible at a time because it does not take into account the visual verification time. These results motivated the extension of the ABILITY MODELER, described above in Section 5.2, to include the explicit model of list selection performance. The results of the summative user study presented in Section 5.6 suggest that this limitation is primarily responsible for the observed shortcomings.

Observations of individual participants

The performance of HM01 (the participant with incomplete tetraplegia using the Head Mouse) improved by up to 43% with respect to the baseline when using the personalized GUIs. His main difficulty in operating the baseline interface was caused by combo boxes which caused long lists with scroll bars to drop down – any accidental clicks while operating those lists would cause the combo box to collapse and force the participant to start over with that widget.

The particular software setup used by ET01 (the able-bodied eye-tracker user) made it impossible for her to perform drag operations. This made the baseline GUIs unusable for

this participant because they included sliders which, in the particular Swing implementation used, could not be set to a desired value with just a single click.

For both sets of interfaces, this participant preferred the personalized interfaces over the alternatives. However, she performed the slowest with those interfaces. The personalized interfaces included list widgets that showed fewer values at a time than the alternative GUIs. The time to manipulate those lists was much longer in practice than what the version of the model used in this pilot predicted. For larger target sizes, both the heights and the widths of list cells were enlarged thus moving the values away from the slider toward the left of the list. That made it much harder for this participant to use her peripheral vision to monitor values as she was using her gaze to click on the slider. She suggested that in some cases larger fonts for list contents would be beneficial because peripheral vision is naturally less acute.

TB01 (trackball, incomplete tetraplegia) was often able to work quickly with GUIs containing small targets, but he tired more quickly when using them and had to take longer breaks compared to when he was using GUIs with larger targets.

M03 (mouse, impaired dexterity) perceived the slider to be difficult to use and in the small condition ranked the VJ02 interface, which used it, as least easy to use even though she was most efficient using it. She was most satisfied with the ET01 interface, which used the largest targets, but which also required more scrolling to operate the lists and subsequently was second slowest to use. The baseline interface was the only one not to use tabs but despite being the easiest to navigate, it was slowest to use for this participant.

VJ02 (VocalJoystic, able-bodied) perceived the preference-optimized GUI as most aesthetically pleasing and most familiar looking and said that this probably made him perceive it as more usable than it perhaps was in practice.

5.6 Summative User Study

In the rest of the chapter, I report on a user study evaluating my two approaches for automatically adapting SUPPLE's user interface generation to the users' individual needs. The first approach uses ARNAULD (Chapter 4) to adapt the interface generation to a user's subjective preferences. The second approach uses the ABILITY MODELER developed in this

Figure 5.7: Different strategies employed by our participants to control their pointing devices (MI02 uses his chin).

chapter to model a user's objective motor performance. In this study, I investigate the robustness and ease of use of the two personalization mechanisms, as well as the quality of the resulting interfaces compared to the manufacturers' defaults.

I 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, I evaluated participants' performance and satisfaction when using 9 different user interfaces: 3 were baselines copied from existing software, three were automatically generated for each participant based on his or her preferences, and three were generated based on the participant's abilities.

5.6.1 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 5.4) and they used a variety of approaches to control their pointing devices (Figure 5.7). 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 (Table 5.5).

5.6.2 Apparatus

I used an Apple MacBook Pro (2.33GHz, 3Gb RAM) for all parts of the study. Most participants were tested at our lab using an external Dell UltraSharp 24" display running

Table 5.4: 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).

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 5.5: Numbers of participants with motor impairments depending on a computer for different activities.

Do you rely on being able to use a computer for...	*# out of 11*
Staying in touch with friends, family or members of your community?	10
School or independent learning?	7
Work?	6
Entertainment?	11
Shopping, banking, paying bills or accessing government services?	10

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, I used the 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, I 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 5.4). 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.

5.7 Eliciting Personal Models

In the first part of the study, I used ARNAULD to build a model of participants' preferences and ABILITY MODELER to model their motor abilities.

5.7.1 Method

Preference Elicitation Tasks

I used ARNAULD (Chapter 4) to elicit participants' preferences regarding presentation of graphical user interfaces. Recall that 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 4.2). The queries are 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 application (which included controls for three dimmable lights, overhead projector with selectable inputs, a motorized screen and a ventilation system; see Figure 3.5) and a stereo controller application (with master volume control, CD player, tape deck and a tuner; see Figure 3.19). These applications were unrelated to those used in the next phase of this experiment.

During the subsequent example critiquing phase, the participants were shown what interfaces 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.

Ability Elicitation Tasks

I used the ABILITY MODELER (extended to use and explicitly model list selection tasks) to build a model of each participant's motor abilities. The four types of tasks used to elicit each participant's motor abilities were discussed earlier in this chapter and are illustrated in Figure 5.2. The particular settings used in this study were:

- ***Pointing.*** I 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.*** I varied target size (10–40 pixels, 3 levels), distance (100 or 300 pixels) and direction (up, down, left, right).

- ***List Selection.*** I varied the height of the scroll window (5, 10, or 15 items), the distance in number of items between 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.*** I used 5 targets of diameters varying from 10 to 60 pixels.

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). I then proceeded with preference elicitation followed by ability elicitation, encouraging the participants to rest whenever necessary. At the end of the session, I administered a short questionnaire asking participants to asses how mentally and physically demanding the two elicitation methods were on a 7-point Likert scale, and to state their overall preference.

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. I analyzed subjective Likert scale responses for the main effect of elicitation method using ordinal logistic regression [150].

5.7.2 Results

Subjective Ratings

On a Likert scale (1–7) for how *mentally* demanding (1 = not demanding, 7 = very demanding) the two tasks were, participants ranked ability elicitation as a little more mentally demanding (2.82) than preference elicitation (2.24), but the difference was not significant ($\chi^2_{(1,N=34)}$=1.62, n.s). They did see ability elicitation as much more *physically* demanding (4.73) than the other method (1.82) and this difference was significant ($\chi^2_{(1,N=34)}$=51.23, $p < .0001$).

When asked which of the two personalization approaches they would prefer if they had to choose one (assuming equivalent results), 9 of 11 motor-impaired participants preferred the preference elicitation (6 strongly) quoting lower physical demand as the primary reason. The two motor impaired participants who somewhat preferred ability-elicitation commented that it felt like a game.

Among able-bodied participants, 3 strongly preferred preference elicitation, 2 somewhat preferred ability-elicitation, and 1 had no preference for either approach.

Preference Model

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 par-

ticipants or the system alone, it does offer support that the resulting interfaces will reflect users' stated preferences accurately.

5.8 Experiment

In this section, I describe the main experiment that evaluated the effects on performance and satisfaction of automatically generated personalized interfaces compared to the baseline versions of those interfaces. Both types of personalized interfaces were tested: those based on participants' stated preferences and those based on their measured abilities.

5.8.1 Method

Participants and Apparatus

The same participants took part in both phases of the study, using the same equipment configurations.

Tasks

I 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. The synthetic application was used because it exhibits 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).

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 5.8). For the synthetic application, I used the baseline from the pilot study (Figure 5.10 left) because it has a very "typical" design for a dialog box: it is compact, and relatively uncluttered.

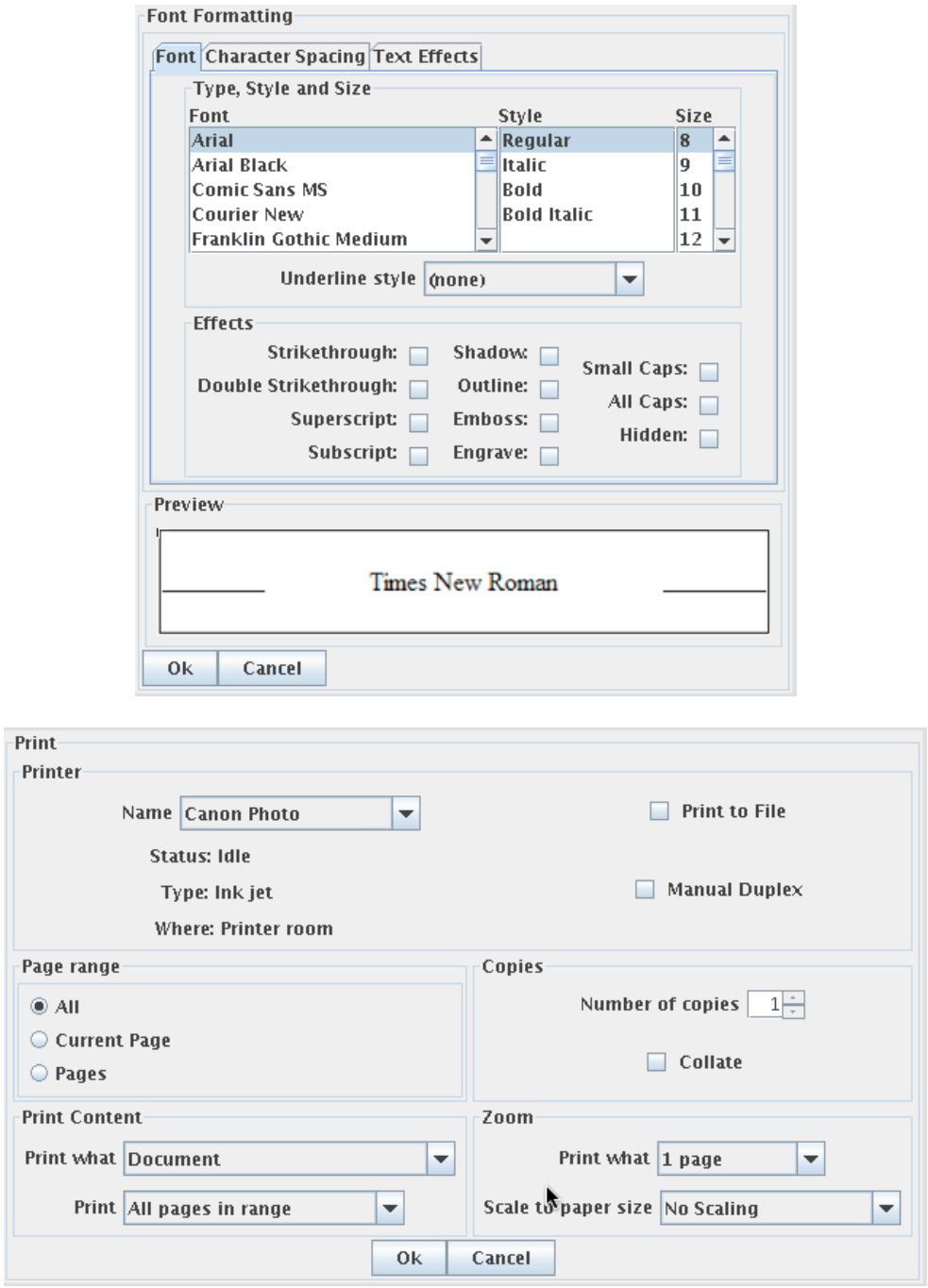


Figure 5.8: The baseline variant for the font formatting and print dialog boxes. They were designed to resemble the implementations in MS Office XP. Two color selection widgets in the font formatting interface were removed and the preview pane was not functional.

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

For the automatically generated user interfaces, I set a space constraint of 750×800 pixels for print and synthetic applications and 850×830 pixels for the font formatting application (see Figures 5.9 and 5.10 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 my particular hardware configurations. I used the same space constraints for all participants to make results comparable.

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 5.8 top), they would have to perform two separate operations: click on the Font tab and then select the Style.

As in the pilot study, during each set of tasks, participants were guided visually through the interface by an animated rectangle (Figure 5.6). An orange border indicated what 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. I chose to use this approach because I was 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.

Procedure

I presented participants with each of the 9 interfaces: 3 applications (font formatting, print dialog, and synthetic) × 3 interface variants (baseline, preference-based, and ability-based) in turn. 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 for any of the 3 applications were identical and were presented in the same order. I 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, I administered a short questionnaire, asking them to rate the variant's usability and aesthetics. After each block of three variants for an application, I additionally asked participants to rank the three interfaces on efficiency of use and overall preference. Finally, at the end of the study, I administered one more questionnaire recording information about participants' overall computer experience, typical computer input devices used, and their impairment (if any).

Generated Interfaces

Figure 5.9 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 5.10) tended to be very diverse, as each participant had different assumptions about what interfaces would be easier to use for him or her.

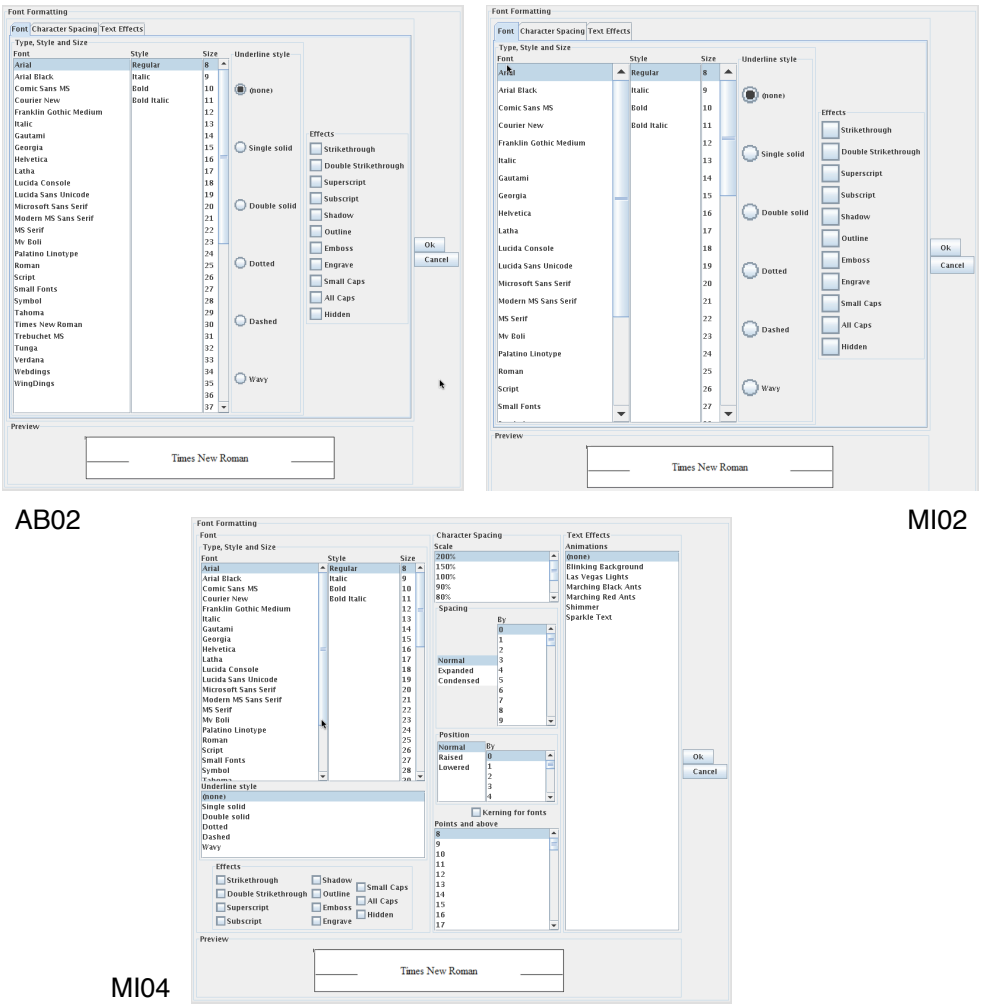


Figure 5.9: User interfaces automatically generated by SUPPLE for the font formatting dialog based on three users' individual motor abilities. The interface generated for AB02 was typical for most able-bodied participants: small targets and tabs allow individual lists to be longer, often eliminating 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 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.

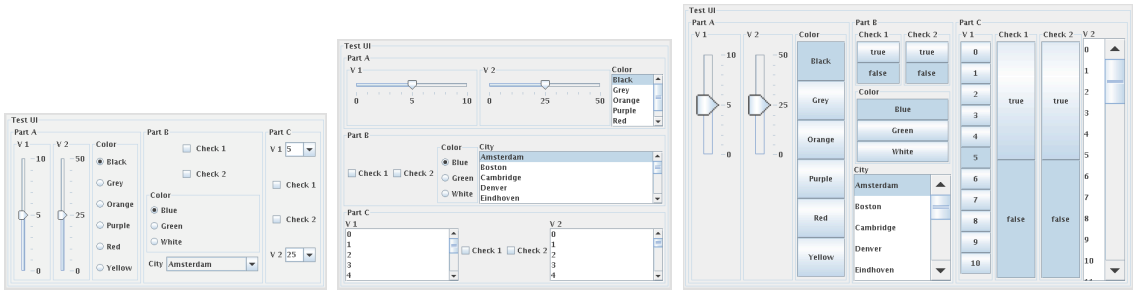


Figure 5.10: User interfaces for the synthetic application. The baseline interface is shown in comparison to interfaces generated automatically by SUPPLE based on two participants' preferences. Able-bodied participants like AB03 preferred lists to combo boxes but preferred them to be short; all able-bodied 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.

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:

- **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, 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 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; I regarded "errors" as any clicks that were not part of setting the target widget to the correct value.

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

For analysis, I took the logarithm of all timing data to adjust for non-normal distributions, which are often found in such data [4]. I 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 [82, 126]. In

these results, I 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, I analyzed error rates as count data using regression with an exponential distribution [144]. Subjective Likert scale responses were analyzed with ordinal logistic regression [150] 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 [63].

5.8.2 Results

Adjustment of Data

I 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.

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 interface variants. As illustrated in Figure 5.11 (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, $p < .0001$).

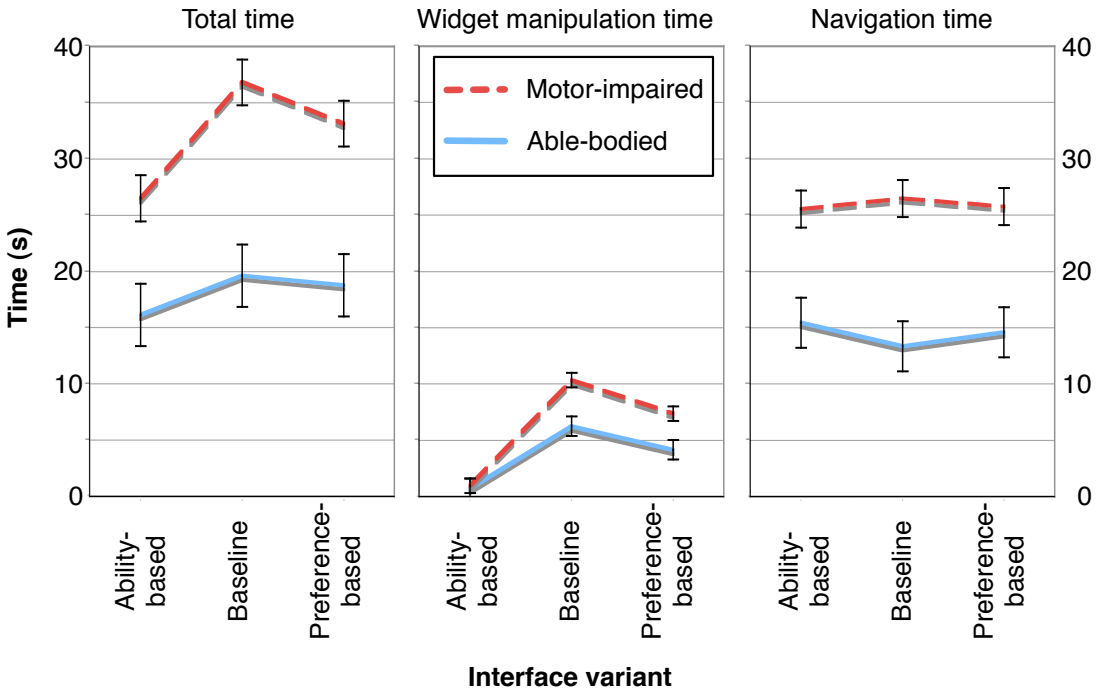


Figure 5.11: Participant completion times. Both motor-impaired and able-bodied participants were fastest with the ability-based interfaces. The baseline interfaces were slowest to use. Error bars show standard error.

The differences in performance can be explained by a significant[2] 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.

[2]The manipulation time data had bi-modal distribution because for many task sets the total manipulation time was 0. I therefore used a non-parametric Wilcoxon Rank Sum test [148] to analyze these data.

I 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 20.5, 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.

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 the effects were similar and significant ($\chi^2_{(2,N=54)}$=23.66, $p < .0001$ for able-bodied and $\chi^2_{(2,N=99)}$=11.00, $p < .01$ for motor-impaired) for both groups individually (Figure 5.12).

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.

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-

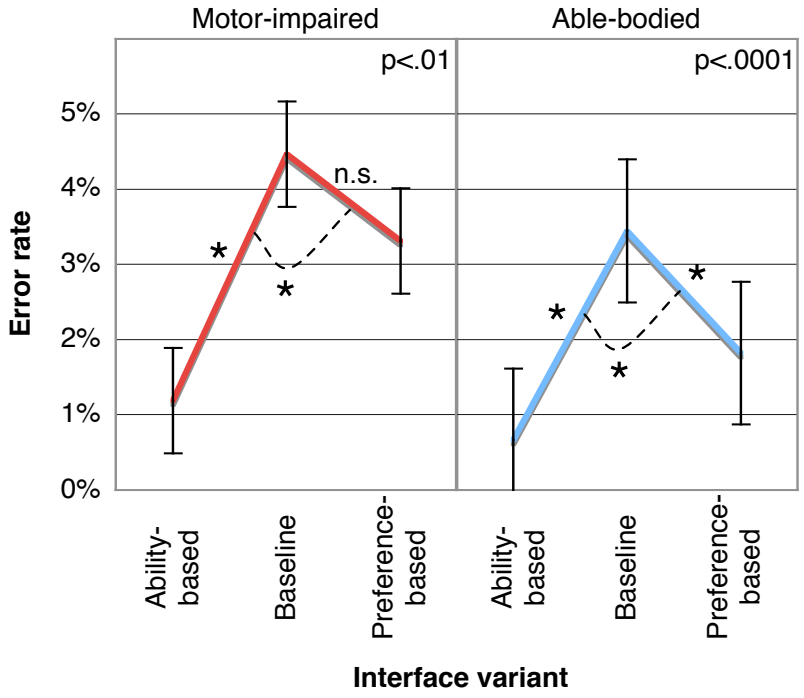


Figure 5.12: Participant error rates. Both motor-impaired and able-bodied participants made fewest errors with the ability-based interfaces. The baseline interfaces resulted in most errors. Error bars show standard error. Significant pairwise differences are indicated with a star (*).

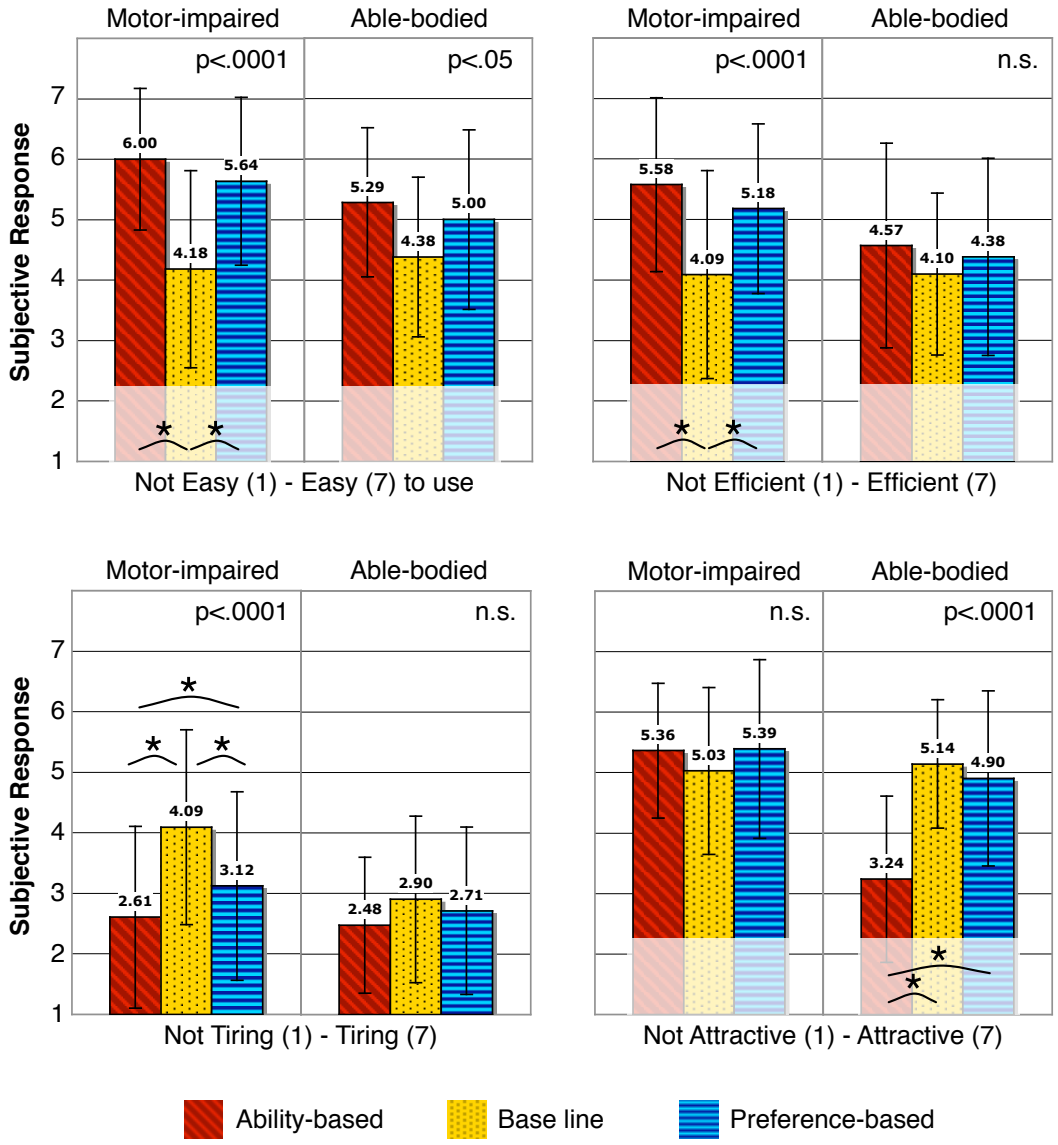


Figure 5.13: Subjective results. Both groups of participants found ability-based interfaces easiest to use. Motor-impaired participants also felt that they were most efficient and least tiring. Able-bodied participants found ability-based interfaces least attractive but, interestingly, motor-impaired 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 (∗).

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 5.13, 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 exist between the baseline condition and each of the automatically-generated ones for participants with motor impairments only.

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 were significant for participants with motor impairments for this measure.

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 for each application by efficiency of use and overall preference (Table 5.6), 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, $p < .001$).

Table 5.6: Average subjective ranking by efficiency and overall preference (1=best, 3=worst)

	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
OverallRank	1.64	2.48	1.88	1.95	2.00	2.05

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.

Participant Comments

MI01, whose dexterity started to deteriorate only recently, commented that the baseline interfaces would be what she had preferred just a few years earlier, but now she found both kinds of the personalized interfaces preferable. MI02, who controls a trackball with his chin and types with a head-mounted wand, said that he uses the trackball only about 20% of the time when manipulating GUIs and the rest of the time he uses the keyboard because despite being slow, it is easier. If more interfaces were like the ability-based interfaces in the study, he said he would use the trackball more often.

MI06 observed that many widgets have pretty large clickable areas but that it is hard to tell that they are indeed clickable (e.g., labels next to radio buttons, white spaces after list items) and that clear visual feedback should be given when the mouse pointer enters such an area. Indeed, the impact of visual feedback on performance has been documented by others [5], and I also observed that many of the motor-impaired participants were very "risk-averse" in that they carefully moved the pointer to the center of the widget before clicking it, which they perhaps would not do if they could be sure that a click elsewhere would be effective.

5.9 Discussion

The participants with motor impairments were significantly faster, made many fewer errors, and strongly preferred automatically-generated personalized interfaces over the baselines.

The results were particularly strong and consistent for ability-based interfaces adapted to their actual motor capabilities using models generated by the Ability Modeler: participants were between 8.4% and 42.2% (mean=26.4%) faster with those interfaces than with the baselines, they preferred those interfaces to all others, and they found those interfaces the easiest to use, the most efficient, and least physically tiring. 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 current difference in performance between users with motor impairments and able-bodied users is at least partially due to user interfaces being designed with a particular set of assumptions in mind—assumptions that are inaccurate for users with motor impairments. By generating personalized interfaces which reflect these users' unique capabilities, I have shown that it is possible to greatly improve the speed and accuracy of users with motor impairments, even when they use commodity input devices such as mice and trackballs.

These results also confirm that the right trade-offs to make in user interface design depend on a particular user's abilities. Even able-bodied participants were faster and made fewer errors with ability-based interfaces, and even they recognized these interfaces as significantly easier to use than the alternatives. In the end, however, they found the ability-based interfaces—which exchanged sparseness and familiar aesthetics for improved ease of manipulation—to be uglier and generally no more preferable than the baselines.

Particularly striking in this study was the situation of MI02, who was 2.85 times slower than an average able-bodied participant using baseline interfaces, but only 1.99 times slower when using interfaces designed for his unique abilities. MI02 controls the trackball with his chin and types on a keyboard with a head-mounted wand; therefore, keyboard shortcuts are also inconvenient for him. Furthermore, his speech is significantly impaired so he cannot use speech recognition software. He works as an IT consultant so his livelihood is critically dependent on being able to interact with computers effectively. Currently, he has to compensate with perseverance and long hours for the mismatch between the current user interface designs and his abilities. He was the slowest participant in this study, but with

the ability-based user interfaces generated by SUPPLE, he was able to close nearly half the performance gap between himself and able-bodied participants using baseline interfaces.

Chapter 6

THE DESIGN SPACE OF ADAPTIVE USER INTERFACES

In the earlier chapters, I developed methods for adapting user interfaces to users' devices, long-term usage patterns, preferences and abilities. All of these adaptations were aimed at properties that change slowly, if at all, over time. In contrast, users often use the same software for a variety of tasks, and those tasks can change frequently. Different tasks may emphasize different subsets of the application's functionality. For this reason, in this chapter, I explore methods for ongoing adaptation of user interfaces for the current task at hand. The term "adaptive user interfaces" is used by different authors to describe very different adaptation approaches. In this chapter, I concentrate on user interfaces where the *presentation and organization of functionality* is adapted over time. Examples of commercially deployed adaptive interfaces of this type include, Microsoft's Smart Menus™ and the Windows XP Start Menu.

Despite considerable debate, however, automatic adaptation of user interfaces remains a contentious area. Proponents of adaptation (e.g., [9, 88]) argue that it offers the potential to optimize interactions for an individual user's typical tasks and style. Critics (e.g., [32, 132]) maintain that the inherent unpredictability of adaptive interfaces may disorient the user, causing more harm than good.

Surprisingly, there appear to be few experimental results that systematically evaluate the space of adaptive designs in a manner which informs the discussion. The few studies that have been published (see Section 2.5) show examples of both successful and unsuccessful adaptation methods, but do not comprehensively consider the reasons underlying this success or failure, even when they report contradictory results on the same adaptive strategy (e.g., two studies of Split Menus: [129] and [32]).

I believe that it is important to understand the reasons that make some adaptive interfaces effective and pleasing to use while others are a frustrating impediment. Empirical and

theoretical studies of these issues are particularly important, because adaptive interfaces are now being introduced into mainstream productivity software and are used by an increasing number of people.

Unlike previous work, I explore several different designs, yielding mixed results and demonstrating that adaptive user interfaces are neither fundamentally good nor bad, but that their success depends on a large number of design decisions as well as the context of use.

In the first experiment, I investigate how three very different adaptive GUIs, which I refer to as the Split Interface, the Moving Interface and the Visual Popout Interface, impact users' satisfaction. For this study, I chose three realistic types of tasks to help improve the external validity of my findings.

I then take the two best adaptations designs and, in the second experiment, study how those designs impact users' performance.

The results of these two experiments consistently showed that one approach to adaptation, which I term Split Interfaces, results in a significant improvement in both performance and satisfaction compared to the non-adaptive baselines. In Split Interfaces, frequently-used functionality is copied to a specially designated adaptive part of the interface while the main part of the interface remains unchanged (see Figure 6.1a).

Therefore, we[1] designed the last experiment entirely in the context of Split Interfaces. In this experiment, we explore a particular design trade-off, namely the relative effects of predictability and accuracy in the usability of adaptive interfaces. I use the term *accuracy* to refer to the percentage of time that the necessary user interface elements are contained in the adaptive area. I say that an adaptive algorithm is *predictable* if it follows a strategy users can easily model in their heads.

I conclude by analyzing my as well as past results and point out those design choices, adaptive algorithm properties, and contextual factors, which clearly affect the success of different adaptive interfaces.

---

[1] I designed, conducted and analyzed the data from this experiment in collaboration with Katherine Everitt [45]

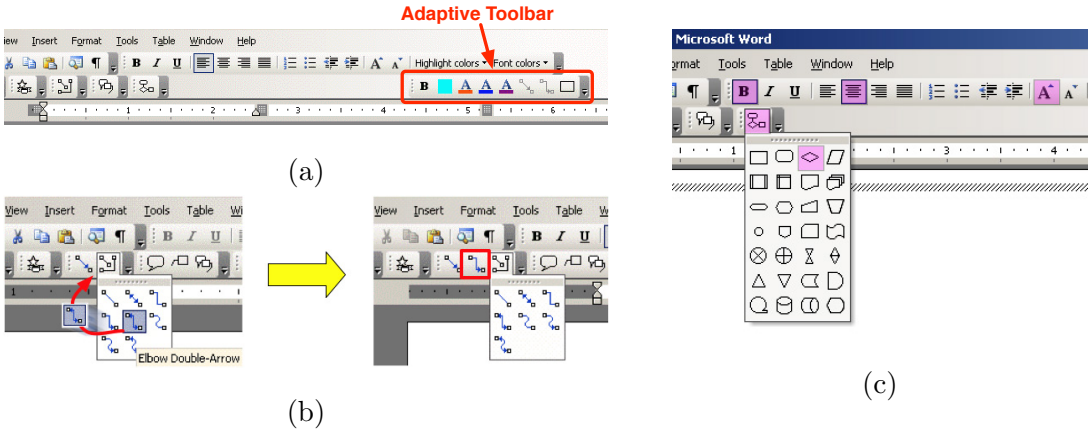


Figure 6.1: Three adaptive interfaces tested in my experiments (as implemented for Microsoft Word): (a) The Split Interface copies frequently used functionality onto a designated adaptive toolbar; (b) The Moving Interface *moves* frequently used functionality from inside a popup menu to a top level toolbar; (c) The Visual Popout Interface makes frequently used functionality more visually salient.

6.1 Adaptation Strategies Tested

I built all of the interfaces using the .NET framework on top of Microsoft Word XP. This allowed me to explore relatively realistic tasks of varied complexity situated in a moderately complex user interface. In choosing the adaptive interface strategies to test, I was looking for approaches that were realistic and that also covered a range of trade-offs between the potential for saving time in manipulating the user interface and the additional cognitive cost. In this section, I describe the specific interfaces I tested.

Non-Adaptive Baseline

In my non-adaptive interface, I ensured that all the toolbars were wide enough to display all of the buttons at all times and no adaptive behavior was presented to the user.

Split Interface (extra toolbar)

I implemented a Split Interface for Microsoft Word by including an additional toolbar (Figure 6.1a). This design is based on the Split Menus interface [129]. But unlike in the Split Menus, where the promoted functionality was *moved* from the original location to the top of the menu, the Split Interface *copies* important functions onto the adaptive toolbar allowing users to either continue using the (unmodified) original interface or to take advantage of the adaptation. Note that the functionality copied to the adaptive toolbar was originally inside pull-down menu panes as well as the already accessible buttons from the top-level toolbars.

If the adaptive toolbar grows too large (8 buttons in my experiments), functionality is demoted to make space for new promotions.

I expected this design to offer users moderate savings with respect to manipulation time while only minimally increasing the cognitive effort required to use the interface.

Moving Interface

Inspired by Sears' concept of moving functionality [129], my Moving Interface moves promoted functionality from inside popup panes onto the main toolbar, causing the remaining elements in the popup pane to shift and also causing the existing buttons on the toolbar to shift to make space for the promoted button (see Figure 6.1b). If there are too many buttons already promoted (8 in the first experiment and 4 in the second) on any given row of toolbars, a new promotion will demote some other button, returning it to its original location.

Unlike in the Split Interface, all elements promoted by this adaptation come from inside popup panes. Thus, I expected it to offer higher advantage with respect to the manipulation time than the Split Interface but at a cost of higher cognitive effort demanded of the users.

Visual Popout Interface

My Visual Popout Interface behaves differently still: it highlights promoted buttons in magenta. If a promoted button resides inside a popup menu, both the button invoking the

popup menu and the menu item are highlighted as shown in Figure 6.1c. In keeping with the design choices I made for the other two designs, in my study, no more than 8 buttons may be highlighted at one time.

This interface clearly does not affect the amount of physical effort involved in manipulating it. I expected it to reduce the visual search time required to find the relevant functionality while not requiring much additional cognitive effort.

Adaptation Algorithms: Frequency and Predictability

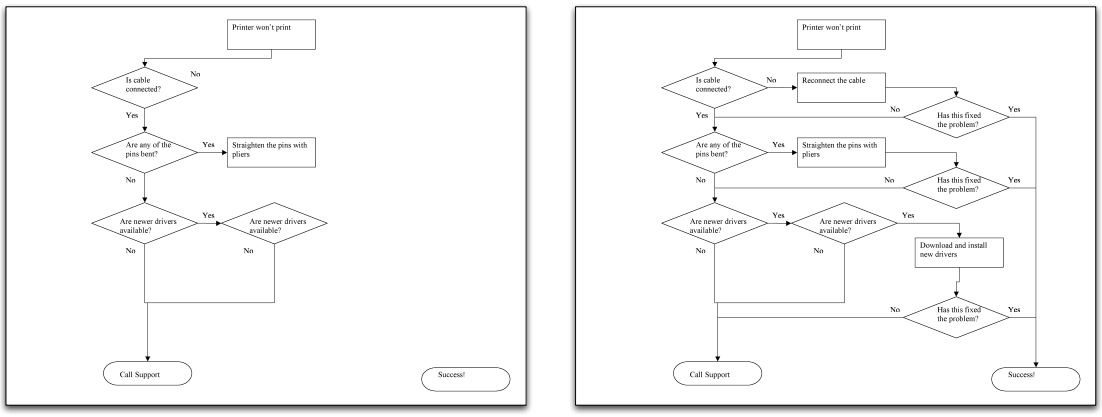
I used two algorithms for choosing what functionality to promote within each of the the adaptation approaches. In my *recency-based* algorithm, the N most recently used commands were promoted by the adaptive interface. In my *frequency-based* algorithm, the algorithm computed the most frequently used commands over a short window of interactions (about 20). The latter mechanism resulted in the interfaces adapting a little less frequently (and perhaps less predictably) than the former, although both adapted in a continuous manner.

6.2 Experiment 1: Measuring Subjective Reactions to Adaptation

In the first experiment, I focused on comparing participants' *subjective* responses to the three adaptive interfaces and the non-adaptive baseline version within Microsoft Word. The tasks used in this study were of medium complexity, chosen to mimic real-world activities for high external validity. I designed the tasks to be engaging enough that participants did not mind repeating similar versions of them across the different user interface types during a session.

6.2.1 Participants

Twenty-six volunteers (10 female) aged 25 to 55 (M=46 years) from the local community participated in this study. All participants had moderate to high experience using computers and were intermediate to expert users of Microsoft Office-style applications, as indicated through a validated screener. Volunteers received software gratuities for participating.



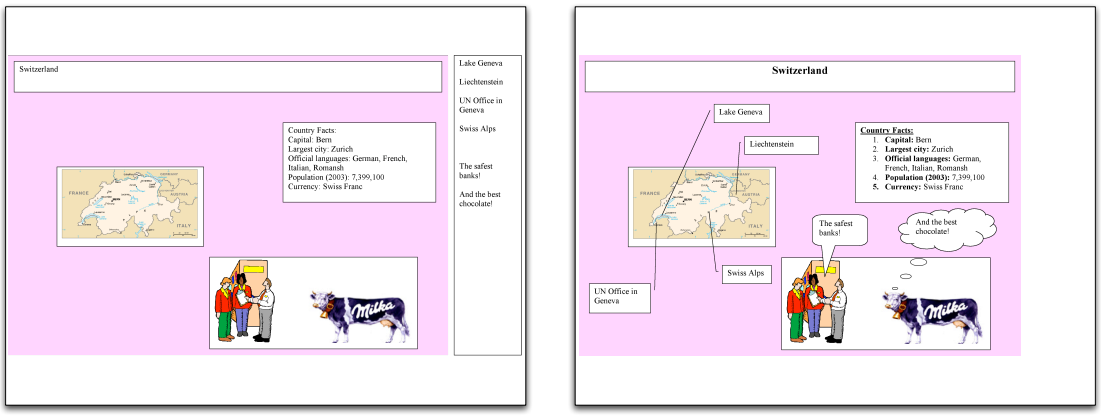
start goal

Figure 6.2: Example of the flowchart task: (left) participants started with the incomplete version and (right) were asked to draw remaining shapes and connectors basing their work on a printed copy of the complete diagram. Participants did not have to enter the missing text.

6.2.2 Tasks

Flowchart

In the flowchart task illustrated in Figure 6.2, participants completed a flowchart of a troubleshooting procedure that was purposefully missing key aspects from its design. Each of these flowcharts, when completed, had 13 components plus connecting arrows. I taught participants how to use the toolbars in Word and provided them with a printout of the completed flowchart they were trying to reproduce. I instructed them to add all of the 16 missing parts but not to spend time aligning the parts precisely as shown on the page because I was interested in their use of the toolbars to add the missing parts and not in the exact alignment of the image. I also instructed them to use the cut tool from the top toolbar if they needed to delete anything they might have accidentally added. In order to keep all of the flowchart tasks isomorphic, participants had to add the same number and kinds of elements to each (3 diamond shapes, 2 rectangles and 11 arrows). No text had to be added.



start goal

Figure 6.3: Example of the poster task: (left) participants started with the incomplete version and (right) were asked to draw remaining shapes basing their work on a printed copy of the complete poster. Any text that needed adding was already available on the right margin and participants simply had to cut and paste it to the correct locations.

Quotes

In the quotes task, I presented participants with a table in Word that showed 24 quotes on 4 different topics, where each topic was assigned its own column. Quotes averaged 2 sentences each. I asked participants to edit the table in the following manner: 6 quotes were in the wrong column and participants were to highlight those misplaced quotes with the corresponding color of the correct column header for that topic; additionally, the font color of the name of each author needed to be highlighted according to the author's birth date (split into four time periods: before 1 B.C., 1 A.D.-1799, 1800-1899, and 1900-1999).

Country Poster

For this task, I presented participants with a one-page, 8.5×11 inch, draft poster in Word summarizing information about a country (see Figure 6.3). They had to edit the poster based on their viewing of the completed poster in their instruction packets. In order to maintain task equivalence, all poster tasks required that participants bold, center and en-

large the font of the title (by clicking on the Enlarge Font button twice), underline, number, and italicize or bold each of 5 facts, add 4 line callouts to the map from toolbars, add a rounded and a cloud callout to each of the two people in the poster, and drag 6 pieces of text from a sidebar into 6 callouts in the poster.

6.2.3 Equipment

I conducted the study using two 3.4 GHz Pentium 4 HP PCs with 1G of RAM to support two simultaneous participants. Each machine drove two NEC MultiSync LCD 1880SX displays set at 1280×1024 resolution. Participants used a Compaq keyboard and Microsoft IntelliMouse for input.

6.2.4 Procedure and Design

I ran participants in pairs. At the beginning of the session, I asked participants to fill out a brief questionnaire about their computer usage experience and habits. I gave them instructions introducing the study and demonstrated each task on a large wall-projected screen. They then performed a set of practice tasks (using the non-adaptive interface) equivalent to those that would be presented in the main part of the experiment. Once they successfully completed the practice task (approximately 10 minutes), I began the test tasks.

The study was a 4 (user interface type: no adaptation, Split, Moving, or Visual Popout) $\times$ 2 (adaptation model: frequency or recency-based algorithm) $\times$ 3 (task: flowchart, quotes, or country poster) design, with user interface type and task as within-subjects factors and adaptation as a between-subjects factor. During the main part of the experiment, participants performed four isomorphic sets of three tasks, each time with a different interface.

I counterbalanced the presentation order of user interface type and adaptation model using a Latin square design across participants. In each pair, participants completed the Flowchart, Quotes, and Poster tasks in the same order—with each of the interfaces that exhibited the same adaptation type but using different underlying adaptation algorithms (i.e., each participant in the pair used a different adaptation model but the same interface

look and feel). Between sessions using different interfaces, I explained how to use the next interface. Because the task sets for each task were isomorphic and the tasks relatively distinct from each other, I expected that interactions across conditions would be more important than task ordering effects. Hence, the order of tasks was kept constant for each user interface condition.

After each of the 4 task series with one user interface, the participants were asked to fill out a brief questionnaire. After the last user interface condition, participants ranked the four user interface types and explained their first and last choices. Finally, the participants were debriefed. The sessions lasted about 2 hours each.

6.2.5 Measures

Dependent variables collected included participant satisfaction ratings, overall preferences, and task times. As expected, there was no effect of interface type on the task times.

6.2.6 Results

Satisfaction

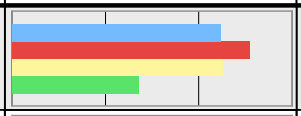
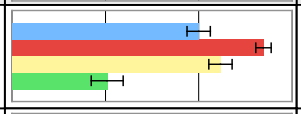
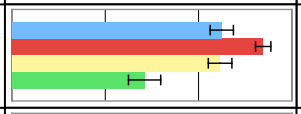
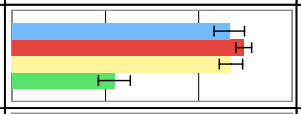
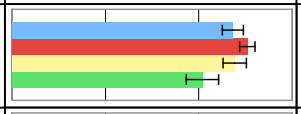
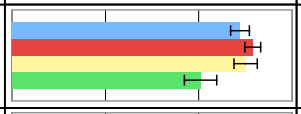
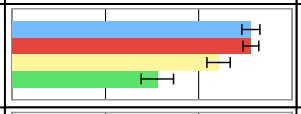
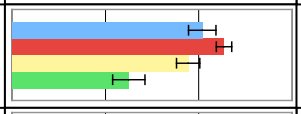
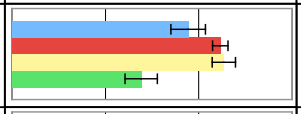
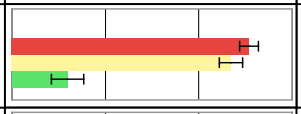
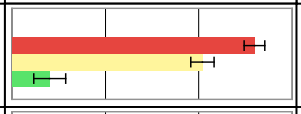
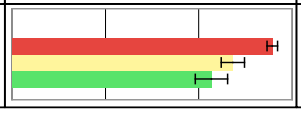
I used ordinal logistic regression [150] to analyze the main effects of user interface type on subjective responses. Logistic regression is non-parametric test well suited to ordinal data where one cannot justify any assumptions about the distribution underlying the data.

For the subjective responses that resulted in a statistically significant effect, I additionally conducted a Wilcoxon Signed Rank test (with Holm's sequential Bonferroni procedure [130]) for pairwise differences. All results are summarized in Table 6.1.

There was a main effect of user interface type on the combined satisfaction ratings ($\chi^2_{(3,N=1066)} = 262.7, p < .0001$) explained by Split Interface being generally rated significantly higher than all others, and the Visual Popout being rated significantly lower than all others. There was no significant difference between the Moving and Baseline interfaces.

I also analyzed the final rankings of the four user interfaces using the Friedman non-parametric test and found a significant preference for the Split Interface, $\chi^2_{(3,N=104)} = 48.4, p < .001$.

Table 6.1: Summary of the subjective responses from Experiment 1. Error bars correspond to standard error.

Question	Results - a graphical representation (1 3 5 7)	Baseline	Split	Moving	Visual Popout	Statistically significant pairwise differences
Average Subjective Response $\chi^2_{(3,N=1066)}=263, p < .0001$		5.47	6.08	5.51	3.71	S > B M > VP
Satisfaction $\chi^2_{(3,N=104)}=55.9, p < .0001$		5.00	6.38	5.46	3.04	S > B M > VP
Ease of Use $\chi^2_{(3,N=104)}=34.4, p < .0001$		5.5	6.38	5.46	3.85	S > B M > VP
Frustration (7 = NOT frustrated) $\chi^2_{(3,N=104)}=36.2, p < .0001$		5.65	5.96	5.69	3.19	B S M > VP
Subjective Performance $\chi^2_{(3,N=104)}=8.3, n.s.$		5.73	6.04	5.77	5.08	*N/A*
Discoverability of Features $\chi^2_{(3,N=104)}=7.5, n.s.$		5.88	6.15	6.00	5.04	*N/A*
Sense of Being in Control $\chi^2_{(3,N=104)}=21.2, p < .0001$		6.12	6.12	5.42	4.12	B S > VP
Mental Demand 7 = LOW demand $\chi^2_{(3,N=104)}=22.0, p < .0001$		5.08	5.54	4.77	3.50	B S M > VP
Physical Demand 7 = LOW demand $\chi^2_{(3,N=104)}=18.5, p < .0003$		4.77	5.46	5.54	3.77	S M > B > VP
Adaptation Helped Efficiency $\chi^2_{(2,N=78)}=53.4, p < .0001$		--	6.08	5.69	2.19	S M > VP
Confusion due to Adaptation 7 = LOW confusion $\chi^2_{(2,N=78)}=47.9, p < .0001$		--	6.19	5.08	1.81	S > M > VP
Predictability of Adaptive Behavior $\chi^2_{(2,N=78)}=10.9, p < .0043$		--	6.58	5.73	5.27	S > M VP

I further analyzed each of the 11 responses separately (the results are summarized in Figure 6.1). To preclude any effects that might have appeared purely by chance, I applied the Bonferroni correction [130] and only considered as significant those effects where $p \leq .05/11 = .0045$.

Asked directly about their satisfaction with the four interface types, participants preferred Split Interfaces to all others and were less satisfied with the Visual Popout than with any of the others. They gave similar responses regarding the ease of use. They were also found the Visual Popout more frustrating than the alternatives. However, the participants did not feel that the different interface types affected their performance on the assigned tasks.

Participants did not feel that the different interface types affected their ability to discover critical interface features. But they felt most in control with the baseline and Split interfaces and least in control with Visual Popout. The Moving Interface was not significantly different in that respect from any of the others. In terms of mental demand, participants felt that Visual Popout was more demanding than the other interfaces. Participants also rated it as most physically demanding while they found the two other adaptive interface types—Split and Moving—as least physically demanding.

Concerning just the three adaptive interfaces, participants felt that the adaptation in Split and Moving interfaces helped them be more efficient than the Visual Popout adaptation. They felt that the Split Interface adaptation caused least confusion while Visual Popout was most confusing. Finally, they found the changes in the Split Interface to be more predictable than the changes in either Moving or Visual Popout interfaces.

6.2.7 User Comments

When debriefed, participants confirmed that the tasks did achieve my goal of high external validity, being realistic and engaging. Many participants commented that both Split and Moving interfaces helped them complete the tasks faster. Several participants liked the fact that the Split interface left the original toolbars unchanged, letting the user decide whether or not to take advantage of the adaptation. A few participants also liked the fact that all

functionality related to their current task would end up in one place. Other participants preferred the Moving interface because it put promoted buttons close to their original locations thus letting them discover adaptations opportunistically rather than having to look at the adaptive toolbar. However, some found that same behavior disturbing because it would change the position of other buttons on the toolbars. Not surprisingly, a number of participants also complained about being disoriented when buttons disappeared from popup menus or when the remaining buttons got rearranged. One participant preferred the Visual Popout interface, while others felt that it often made it harder to find what they were looking for by changing the appearance of previously familiar elements, and thus making it harder to find them.

6.2.8 Summary

This first study was important as it allowed me to observe participants interacting with the user interfaces on tasks that had high external validity. Unsurprisingly, the high variability added by the cognitive decisions made within these tasks rendered task times insensitive to the experimental manipulations. In fact, the task-time analyses did not show any significant differences across the 3 tasks with the exception of the Quotes task, and even then, only the Visual Popout condition was observed to be significantly slower than the other 3 conditions, and this was likely due to implementation issues, hurting performance. Because the satisfaction data showed significant preferences for the Split Interface condition over no adaptation and Visual Popout, I assumed participants perceived benefits to this kind of adaptation that time measures for these tasks were not sensitive enough to capture. For this reason, I decided to run a second experiment.

6.3 Experiment 2: Impact of Adaptation on Performance

In the second experiment, I relaxed the external validity requirements by reducing the cognitive complexity of the tasks. I assumed that while the tasks would be less realistic, having the participant press more buttons while making fewer cognitive decisions would allow me to more carefully measure performance differences that might exist between the user interface types. This also provided me with much tighter control over the order of

button presses so that I could examine the effectiveness of the adaptation schemes under different predictive accuracy conditions.

Because I found no preference differences between my frequency and recency-based adaptation models in Experiment 1, I decided to drop this variable from the study. Additionally, because my implementation of the Visual Popout interface was clearly less preferred than the non-adaptive baseline, I dropped that condition from this study as well.

6.3.1 Participants

Eight volunteers (2 female) aged 25 to 58 (M=36 years) participated in this study. All participants had high experience using computers and were expert users of Microsoft Office-style applications. Participants received a small gratuity for participating.

6.3.2 Task and Procedure

At the beginning of each session, I explained and demonstrated the task and each of the user interfaces. After that, the participants worked on a practice task to familiarize themselves with the location of different commands and with the adaptive interfaces. In order to isolate the effects that the various manipulations had on toolbar usage, I used a task—illustrated in Figure 6.4—in which the interface told participants exactly which toolbar buttons they had to press (all of those buttons were located inside popup menus accessible from the toolbars). In this task, the system presented an image of a particular command within the Word document. Each participant had to find and hit this button on the toolbar as quickly and accurately as they could. Then they hit a done button, also presented within the document, and immediately got a new command to target. Each participant repeated this 52 times for each trial, although the first 12 were considered a warm up necessary to initiate (seed) the adaptive models and were not included in the results. Additionally, after each of the three conditions, I administered a brief questionnaire, asking the participants how easy each interface made it to find the functionality and how the participant felt it improved his or her efficiency.

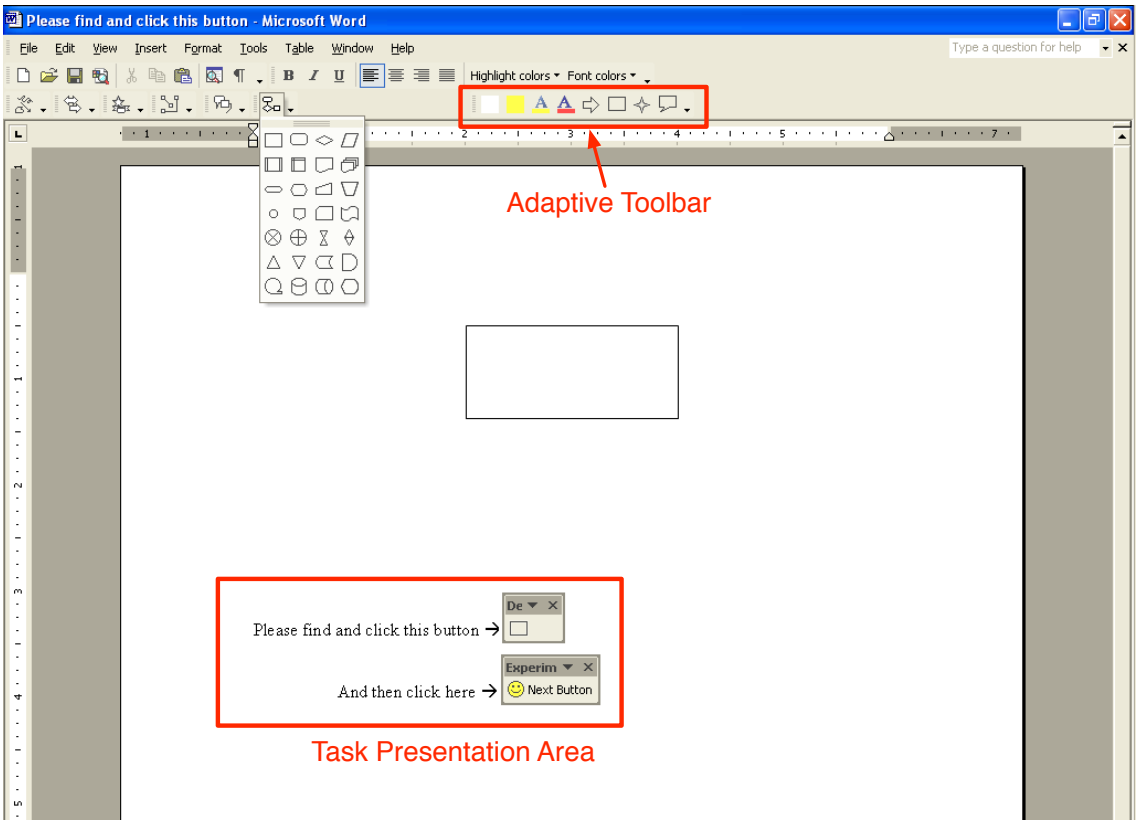


Figure 6.4: Task setup for Experiment 2: The system presented an image of a target button in the task presentation area; participants had to find and click a corresponding button in the interface, and complete a trial by clicking the "Next Button" button in the task presentation area—if correct adaptation took place participants had a choice of clicking on the target the button at its original location or on the one in the adaptive toolbar.

I created two classes of tasks, which resulted in the recency-based adaptive algorithm being either 30% or 70% accurate (i.e., correctly promoting the next button to be clicked by the user) in the case of the two adaptive conditions. This was especially interesting in the Split Interface, where participants could choose either to use the regular toolbar item or the promoted one in the adaptive toolbar. Therefore, the experiment was a 3 (user interface type: no adaptation, Split Interface, or Moving) $\times$ 2 (accuracy: 30% or 70%)

within-subjects design. All conditions were fully counterbalanced to control for the effects of training.

6.3.3 Equipment

I ran the experiment on two 2.8 GHz Pentium 4 Compaq PCs with 2G or RAM with the same input/output devices as used in the previous experiment.

6.3.4 Results

Task Times

I ran a 3 (user interface type) $\times$ 2 (accuracy: 30% or 70%) $\times$ 2 (trial order: 30% first or 70% first) repeated measures ANOVA, with user interface type and accuracy within-subjects, and trial order run between-subjects.

I found a significant main effect of user interface type ($F_{2,12} = 8.545, p < .01$). Pairwise comparisons using a Bonferroni correction revealed that participants were significantly faster using the Split Interface, $p < .01$, and marginally faster using the Moving Interface, $p = .073$, than without adaptation. The two adaptive interfaces were not significantly different from each other.

I also observed a main effect of accuracy ($F_{1,6} = 8.859, p < .05$). I found a significant interaction between user interface type and accuracy ($F_{2,12} = 7.689, p < .01$), driven by both adaptive interfaces resulting in faster performance (Split: $p < .001$, Moving: $p < .001$) with the 70% accuracy scenario faster than with 30% accuracy.

Finally, I saw a significant interaction between accuracy and trial order ($F_{1,6} = 6.515, p < .05$). Participants who saw the 70% condition followed by the 30% condition had a marginal decrease in performance, while participants that saw the 30% first showed vast improvements when they had the 70% condition ($p < .001$). I believe that this is due to various strategies participants built up from using one interface or the other, but verifying this remains future work.

Utilization of the Adaptive Interface

Additionally, planned analyses of the Split Interface usage data showed that the level of accuracy significantly affected the way participants interacted with the Split Interface ($F_{1,6} = 10.361, p < .05$). On average, when functionality existed in both places, participants utilized functionality from the extra toolbar more frequently in the 70% accuracy condition (M=93.1%) than in the 30% condition (M=81.0%).

Satisfaction and User Comments

The analysis of the two subjective responses using ordinal logistic regression did not reveal any significant effect of user interface type on the participants' perception of how easy each interface made it to find the functionality and of how much the interface improved his or her efficiency.

But in their post-experiment comments, participants focused primarily on three issues: ease of discovery, use of the adapted functionality, and the confusion caused by the adaptive interface. Many found the Split Interface not very useful because it required them to look in two distinct places for any one piece of functionality (unlike in the first experiment, they saw no benefit to having frequently used functionality grouped together). The Moving Interface was considered more convenient in that respect. The Moving Interface, however, caused items in pull-down menus to shift and also caused buttons on the toolbars to move horizontally as functionality was promoted or demoted. Even some of the participants who preferred the Moving Interface overall found this to be a concern.

6.4 Experiment 3: Accuracy versus Predictability

In this experiment I explore the trade-off between predictability and accuracy in the usability of adaptive interfaces. As in Experiment 2, I use the term "accuracy" to refer to the percentage of time that the necessary interface elements are contained in the adaptive area. I say that an adaptive algorithm is "predictable" if it follows a strategy users can easily model in their heads. I focus on these properties because they reflect a common design trade-off in adaptive user interfaces: whether to use a simple, easily-understood strategy

to promote functionality, or whether to rely on a potentially more accurate but also more opaque machine learning approach?

6.4.1 Hypotheses

Building on the first two experiments and on previous research, we formulated the following hypotheses when designing this study:

1. The higher the accuracy of the adaptive algorithm, the better the task performance, utilization, and the satisfaction ratings.

2. The more predictable the adaptive algorithm, the better the task performance, utilization, and the satisfaction ratings.

3. Increased predictability would have a greater effect on satisfaction and utilization than increased accuracy. We were led to this hypothesis by the design heuristic that successful user interfaces should be easy to learn [104].

6.4.2 Participants

Twenty-three volunteers (10 female) aged 21 to 44 (M=35 years) from the local community participated in this study. All participants had moderate to high experience using computers and were intermediate to expert users of Microsoft Office-style applications, as indicated through a well-validated screener. All participants had normal vision, and received a software gratuity for participating.

6.4.3 Task

We used a task setup very similar to that used in Experiment 2 except for the location of the adaptive toolbar. In this experiment, the adaptive toolbar was located in the upper right—far enough from the closest relevant non-adaptive button ($> 20^\circ$ visual angle) so that an explicit change of gaze was necessary to discover if a helpful adaptation had taken place (see Figure 6.5). Eight buttons were always shown in the adaptive toolbar, and at most one button was replaced per interaction. During each task set, participants clicked on

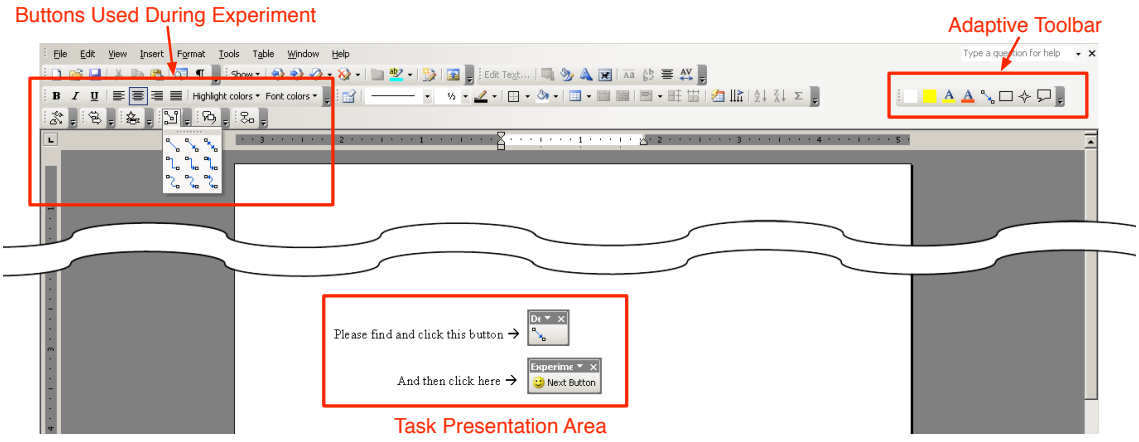


Figure 6.5: Task setup for Experiment 3 was based on the setup for the previous experiment. The main differences were that the adaptive toolbar was placed further apart from the part of the interface utilized during the experiment, and that only buttons inside popup menus were used.

60 target buttons. We considered the first 10 clicks to be a ramp-up time and hence did not include performance metrics for these first clicks in the analysis.

6.4.4 Equipment

As in the two previous experiments, two 2.8 GHz Pentium 4 Compaq PCs were used, each with 2G of RAM. Each computer was equipped with an optical mouse and with two NEC 18" color LCD displays, but only one display per computer was actively used by the participants. Also as before, up to two volunteers participated in the study at a given time.

6.4.5 Design and Procedure

The study considered two accuracy levels: 50% and 70%. Because it is difficult to implicitly measure predictability, and measuring it explicitly might influence user performance [112], we considered two extreme cases: in the unpredictable condition, updates to the adaptive toolbar were entirely random—a worst-case simulation of a complex, machine-learning algorithm's inscrutability. In the predictable condition, we chose a most recently used strategy

(the same as the recency-based algorithm in the first experiment), placing the eight most recently used buttons in the toolbar. Post-experimental interviews confirmed that the participants easily formed a mental model of this policy. Hence, the study was a 2 (accuracy: 50% or 70%) × 2 (predictability: *High* (most recently used) or *Low* (random)) fully counterbalanced design.

We predetermined the sequence of buttons that had to be pressed in all conditions as well as the contents of the adaptive toolbar in the random condition to ensure the desired level of accuracy.

After familiarizing themselves with the task and completing a practice set using a non-adaptive interface, participants performed four counterbalanced task sets, one for each of the four conditions. Participants filled out a brief satisfaction survey after each task set, and an additional survey and an exit interview following the last session. Participants took 2.5 to 5.0 minutes per task set, and the whole study took less than an hour per participant.

6.4.6 Measures

We collected overall task times as well as the median times to acquire individual buttons (i.e., the time from selecting the "Next Button" to clicking on the indicated target), distinguishing times for buttons in their original locations from those located on the adaptive toolbar. We also measured the adaptive toolbar utilization levels, or the number of times that the participant selected the requested interface element from the adaptive toolbar divided by the number of times that the requested element was present on the adaptive toolbar. Additionally, we collected the subjective accuracy of the adaptive algorithm, and participant satisfaction ratings (on a 7-point Likert scale). Finally, a random subset of participants was asked to perform an extra set of tasks following the main experiment; here we used an eye-tracker to determine which strategies the participants employed. Performance considerations prevented me from using the eye tracker during the main part of the experiment.

Table 6.2: Summary of the results for Experiment 3. Times are in seconds, satisfaction ratings are on a 7-point Likert scale, and statistical significance was tested at $p = .05$ level.

		Individual conditions				Averaged over accuracy settings			Averaged over predictability settings		
		Random, 50%	Random, 70%	Predictable, 50%	Predictable, 70%	50%	70%	significant?	Random	Predictable	significant?
Duration		196	176	199	177	197	177	*	186	188	
Utilization		69%	89%	73%	84%	71%	86%	*	79%	78%	
Satisfaction ratings	Useful	3.36	4.77	3.73	4.77	3.55	4.77	*	4.07	4.25	
	Predictable	2.41	3.00	3.82	4.43	3.11	3.71		2.70	4.12	*
	Knew†	1.95	2.64	3.24	4.24	2.60	3.44		2.30	3.74	*
	Frustrating	4.23	2.55	3.50	2.73	3.86	2.64	*	3.39	3.11	
	Confusing	4.14	3.36	3.24	2.95	3.69	3.16		3.75	3.09	
	Satisfied	3.86	4.64	4.41	5.05	4.14	4.84		4.25	4.73	
	In Control	3.19	4.27	4.41	5.05	3.80	4.66	*	3.73	4.73	*
	Efficient	2.59	3.95	3.18	4.59	2.89	4.27	*	3.27	3.89	

†Knew = "I knew when the extra toolbar would have what I needed"

6.4.7 Results

We analyzed all continuous measures using a 2 (50% or 70% accuracy) × 2 (High vs. Low predictability) repeated measures ANOVA. For the analysis, we took the logarithm of all timing data—standard practice to control for non-normal distributions found in such data [4]. Because one cannot justify any assumptions about the distribution underlying Likert scale subjective responses, and because three participants omitted answers to some of the questions, we used ordinal logistic regression [150] (a non-parametric test, which can accommodate missing data) to analyze those data. Subjective data from one participant were lost due to a software error. Table 6.2 summarizes the results.

Perception of Predictability

In the free response part of the post-task questionnaire, 11 out of 23 participants spontaneously commented, after at least one of the two random conditions, that the toolbar behavior was "random," "confusing," or otherwise unpredictable. In contrast, after the

predictable conditions only two participants commented that they did not understand the adaptive toolbar's behavior, while three specifically observed that it behaved more predictably than in earlier conditions.

Similarly, when debriefed after the study, the majority of participants correctly described the algorithm in the predictable condition as selecting the most recently used items, while a few felt that the system selected the most frequently used items. Participants often described the algorithm in the random condition as behaving in an apparently random manner though they often assumed that the behavior was purposeful (even if inscrutable) and that the algorithm was trying to "guess" or "help."

Satisfaction

We observed main effects of predictability ($\chi^2_{(1,N=1049)} = 17.22, p < .0001$) and accuracy ($\chi^2_{(1,N=1049)} = 34.59, p < .0001$) on the combined satisfaction ratings. We further analyzed each of the 8 responses separately. To preclude any effects that might have arisen purely by chance, we applied the Bonferroni correction [130] and considered as significant only those effects where $p \leq .05/8 = .00625$.

This further analysis showed that participants' feeling of being in control increased both with improved predictability ($\chi^2_{(1,N=87)} = 11.69, p < .001$) and with improved accuracy ($\chi^2_{(1,N=87)} = 9.70, p < .002$).

In predictable (vs. random) conditions, participants felt that the adaptive interface behaved more predictably ($\chi^2_{(1,N=87)} = 17.03, p < .0001$), and that they knew better when the adaptive toolbar would contain the needed functionality ($\chi^2_{(1,N=86)} = 15.03, p < .0001$).

In conditions with higher accuracy, participants felt that the adaptive interface was more useful ($\chi^2_{(1,N=88)} = 14.26, p < .001$), less frustrating ($\chi^2_{(1,N=88)} = 26.47, p < .0001$), and that it improved their efficiency ($\chi^2_{(1,N=88)} = 12.56, p < .001$).

Unsurprisingly, a separate 2×2 repeated measures ANOVA revealed that participants perceived a difference in accuracy between the two accuracy levels ($F_{1,19} = 34.906, p < .001$), estimating the lower accuracy condition to be 50.0% and the higher to be 69.0% accurate, on average.

Utilization

A 2×2 repeated measures ANOVA showed a main effect of accuracy on adaptive toolbar utilization ($F_{1,22} = 11.420, p < .01$). At the 50% accuracy level, the adaptive toolbar was used 70.6% of the time when it contained the correct button, compared to 86.4% at the 70% accuracy level.

Performance

The results showed that increased accuracy improved task completion times ($F_{1,18} = 62.038, p < .001$) and, in particular, the median time to access buttons located on the adaptive toolbar (from 1.86s to 1.70s, $F_{1,18} = 18.081, p < .001$) but not those located in the static part of the interface. No significant effects were observed for the algorithm's predictability.

Relative Impacts of Predictability and Accuracy

Treating the condition with low predictability and 50% accuracy as a baseline, we investigated which change would more greatly impact the participants: raising predictability or improving the accuracy to 70%. Thus, we compared two conditions: predictable but only 50% accurate versus random but 70% accurate.

An ordinal logistic regression analysis (with Bonferroni correction) of the satisfaction responses showed that the participants felt that the toolbar in the predictable but 50% accurate condition was more predictable ($\chi^2_{(1,N=44)} = 9.14, p < .003$), while the adaptation in the random but more accurate condition was more useful ($\chi^2_{(1,N=44)} = 11.93, p < .001$), less frustrating ($\chi^2_{(1,N=44)} = 19.07, p < .0001$), and better improved their efficiency ($\chi^2_{(1,N=44)} = 14.80, p < .0001$).

Increased accuracy also resulted in significantly shorter task completion times ($F_{1,22} = 26.771, p < .001$) and higher adaptive toolbar utilization ($F_{1,22} = 5.323, p < .05$) than improved predictability.

Eye Tracking

In analyzing the eye tracking data (22 task sets performed by 16 participants; each condition was repeated 5 or 6 times), we identified three regions of interest (ROIs): the static buttons on the top left, the adaptive toolbar at the top right, and the task presentation area in the center of the screen (see Figure 6.5). The small sample collected led to low statistical power, but the data shed some light on the approaches used by the participants.

We looked at transitions between the ROIs to see if the participants were more likely to look at the adaptive toolbar or the static toolbar after being presented with the next button to click. The results showed that users moved their gaze from the task presentation area to the adaptive toolbar (rather than to the static part of the interface on the left) much less on average in the low accuracy condition than the high one (66% vs. 79%, respectively). The difference between predictable and random conditions did not elicit similar difference in behavior.

We also looked at the percentage of times participants first looked at the adaptive toolbar, failed to find the desired functionality there, and then shifted their gaze to the static toolbar. Here the results showed that participants looked but could not find the appropriate button on the adaptive toolbar much more often in the random than the predictable conditions (41% vs. 34%, respectively). Participants seemed to be performing better than the expected 40% failure rate (averaging over the two accuracy levels) in the predictable condition, suggesting that the more predictable algorithm did help the participants to best direct their effort.

Other User Comments

Besides commenting on the predictability of the adaptive toolbar behavior, 10 of the 23 participants commented that they wished the adaptive toolbar were closer to the original locations of the buttons used in order to aid opportunistic discovery of adaptation. Similar comments were also reported by the participants in the two earlier experiments—the moving interface, although not statistically better than the non-adaptive baseline, was frequently praised for placing adapted functionality right next to the original location. We chose the

Split Interface for this study, because it was statistically better than the baseline, but a hybrid approach might be even better.

6.5 Initial Characterization of the Design Space of Adaptive User Interfaces

Among the three very distinct approaches to adaptation I studied, the Split Interface approach was the only one that resulted in significantly higher satisfaction and performance than the non-adaptive baseline. This is despite the fact that it offered lower *potential* for performance improvement than the Moving Interface, which exclusively promoted hard-to-reach functionality and placed it close to its original location. I attribute Split Interface's success to two related attributes: the high spatial **stability** of this approach and the fact that the use of this type of adaptation is **elective**. The spatial stability can be considered a continuous metric that reflects how much the original interface is altered by the adaptation. If not at all, then the adaptive strategy is elective, in that the user has the choice of whether to take advantage of the adaptation or to ignore it. In contrast, **mandatory** adaptation strategies, like the Moving and Visual Popout interfaces, give the user no choice but to use the adaptive mechanism; they necessarily affect the original user interface resulting in lower stability. The importance of stability and the elective use of the adaptation is further corroborated by the results of Findlater and McGrenere [32] who studied the original design of Split Menus [129], where the promoted items were *moved* rather than copied to the top of the menu. In that laboratory performance study, the Split Menus resulted in lower performance and satisfaction than the non-adaptive baseline.

However, there is a trade-off between user interface stability and the **locality** of the adaptation: a number of users commented positively on the fact that the adaptation in the Moving Interface condition happened very close to the original location of the item, allowing for serendipitous discovery of adaptation. The two performance experiments showed that the utilization of the adaptive toolbar in the Split Interface was substantially lower than 100% suggesting that the lower locality of this adaptive approach did incur some cost.

Despite the general superiority of the Split Interface, **potential effort savings due to adaptation** is an important factor. This factor can be affected by the design of the adaptive interface, as was the case in my studies. Specifically, even though the Moving

Interface adaptations caused more dramatic changes than the Visual Popout Interface, the former resulted in much higher satisfaction ratings because it did offer a noticeable benefit to the users. This factor can also be affected by the context. For example, Findlater and McGrenere [33] demonstrated increased benefits of adaptation on small screen devices, where regular operation of the user interface requires more scrolling than the desktop variant.

The second and third experiments demonstrated the impact of the **accuracy** of the predictive algorithm driving the adaptation: higher accuracy resulted in improved performance, satisfaction, and utilization of the adaptive mechanism, a result that was also observed by others [33, 141]. The third experiment also demonstrated that the **predictability** of the adaptive algorithm can impact users' satisfaction but in my particular experimental setting the predictability did not impact performance. Overall, substantial increases in accuracy appeared to have higher impact on satisfaction and performance than did the changes in predictability. This result is important for analyzing a trade-off in designing adaptive algorithms where one can choose between intuitive approaches (such as promoting most recently used functionality) or potentially more accurate but more opaque machine learning approaches.

Furthermore, Tsandilas and schraefel [141] showed that the impact of accuracy increases with the **cost of incorrect adaptation** in adaptive interfaces employing a mandatory approach. In particular, they studied dynamic menu designs, where promoted items were expanded at the cost of reducing the sizes of other items. In such designs, incorrect prediction of what item the user is going to use next will negatively impact the user's performance when selecting the desired item.

The **frequency of adaptation** appears to have a large impact on the relative weights people assign to the different costs and benefits of adaptation, as illustrated by the conflicting results (both for user satisfaction and performance) of two previous studies of Split Menus ([129] and [32]). The extremely slow pace of adaptation in [129] (once per session) resulted in strongly positive results while the fast pace in [32] (up to once per interaction) caused the same interface to fare worse than the non-adaptive base-line.

My results also indicate that the **pace of interaction** with the interface and the **cognitive complexity of the task** influence what aspects of the adaptive interface users find

relevant. As the differences in user comments between the first two experiments suggest, fast and largely mechanical interactions in the second experiment caused users to pay more attention to the operational properties of the interfaces. In contrast, in the more complex and more slowly-paced interactions of the first experiment, users were less concerned with distance between the extra toolbar and the original location of the adapted buttons but they frequently commented that they appreciated that all relevant functionality was grouped in one place, allowing them concentrate on the task rather than on navigating the interface. This observation should influence how satisfaction data is collected in future studies of adaptive user interfaces. It also suggests that long-term in situ deployment may result in different user feedback than even a realistic laboratory study.

Finally, as the discrepancy between the original results of Greenberg and Witten [51] and the follow up work by Trevellyan and Browne [140] indicate, users' **familiarity with the interface** affects the usefulness of adaptation: as the users approach expert performance and spend less time navigating through the interface, the benefits of certain kinds of adaptation—especially approaches with low spatial stability—will diminish. This observation is further corroborated by the work of Cockburn et al. [28].

6.6 Summary

Motivated by the controversy surrounding user interfaces that automatically adapt to user's behavior, as well as the relative lack of empirical evidence to inform the debate, I conducted three studies to explore different points in the design space of such adaptive user interfaces. Through a discussion of our own and past results, I have identified a number of properties of adaptive user interfaces that are likely to impact the acceptance of such interfaces. In particular, this body of work suggests that Split Interfaces, which duplicate (rather than move) functionality to a convenient place, tend to improve users' performance and satisfaction compared to non-adaptive baselines.

Among other design dimensions, I have examined the influence of accuracy and predictability on adaptive toolbar user interfaces. The results of my particular study show that both predictability and accuracy affect participants' satisfaction but only accuracy had a significant effect on user performance or utilization of the adaptive interface. Contrary to

my expectations, improvement in accuracy had a stronger effect on performance, utilization and some satisfaction ratings than the improvement in predictability. These results suggest that even though machine learning algorithms may produce inscrutable behavior, in certain cases they may have the potential to improve user satisfaction. Specifically, if a machine learning algorithm can more accurately predict a user's next action or parameter value, then it may outperform a more predictable method of selecting adaptive buttons or default values. However, because predictability and accuracy affect different aspects of users' satisfaction, improvements to one of these factors cannot fully offset the losses to the other.

Chapter 7

CONTRIBUTIONS AND FUTURE WORK

The thesis of this dissertation is that automatically generated user interfaces, which are adapted to a person's devices, tasks, preferences, and abilities, can improve users' satisfaction and performance compared to traditional manually designed "one size fits all" interfaces. In support of this thesis, I developed three systems to enable a broad range of personalized adaptive interfaces: SUPPLE, which uses decision-theoretic optimization to automatically generate user interfaces adapted to a person's device and usage; ARNAULD, which allows optimization-based systems to be adapted to users' preferences; and the ABILITY MODELER, which performs a one-time assessment of a person's motor abilities and then automatically builds a model of those abilities, which SUPPLE uses to automatically generate user interfaces adapted to that user's abilities. The results of my laboratory experiments demonstrate that these automatically generated, ability-based user interfaces significantly improve speed, accuracy and satisfaction of users with motor impairments compared to manufacturers' defaults. I also provided the first characterization of the design space of adaptive graphical user interfaces, and demonstrate how such interfaces can significantly improve the quality and efficiency of daily interactions for typical users.

The following section discusses the contributions of this dissertation and Figure 7.1 provides a visual summary.

7.1 Contributions

In Chapter 3, I described 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, an optional usage model reflecting how the interface will be used, and a cost function. SUPPLE naturally generates user interfaces adapted to different devices. It also provides mechanisms for automatic system-driven adaptation to both long-term and

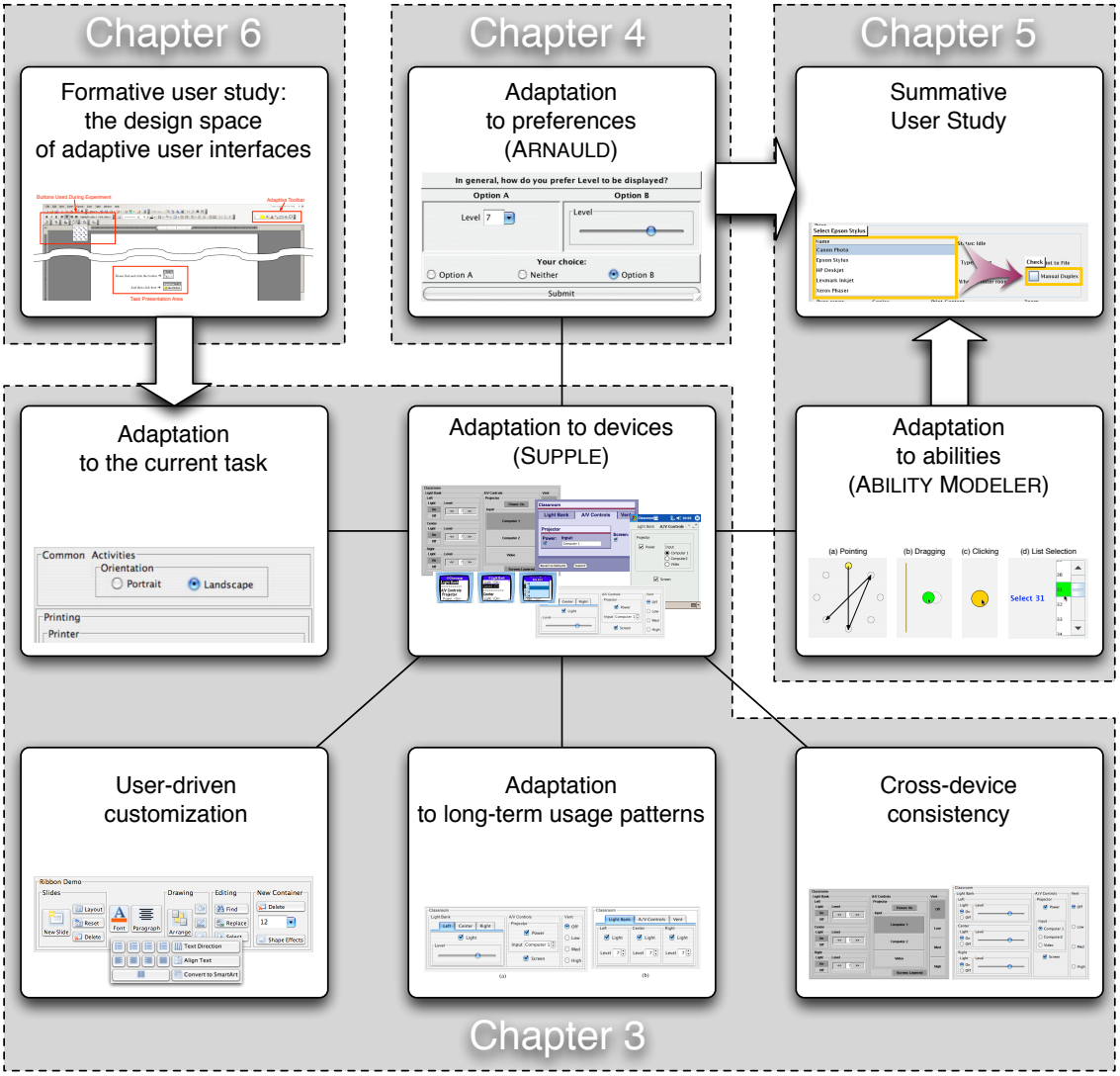


Figure 7.1: A visual summary of the technical contributions of this dissertation.

short-term usage patterns. As a complement to automatic generation and adaptation, SUPPLE also provides an extensive user-driven customization mechanism, which lets users modify any SUPPLE-generated user interface in terms of the presentation of the individual pieces of functionality and the overall organization.

SUPPLE's optimization algorithm can generate user interfaces in under a second in most cases, provided the cost function is expressed in a particular parametrized form. I have also introduced an alternative cost function formulation, which can reflect user's motor capabilities but which results in slower system performance. An important consequence of casting user interface generation as an optimization problems 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 a potential for personalizing the interface generation process and the subsequent two chapters introduced two systems for eliciting users' subjective preferences and objective motor abilities, respectively.

In Chapter 4, I described the ARNAULD system for interactively eliciting user preferences for the purpose of automatically learning parameters of the cost function. ARNAULD lets users specify their preferences by providing simple feedback on examples of concrete user interfaces either through user-driven customization or through system-driven active elicitation interactions. ARNAULD uses a novel max-margin learning algorithm that turns the user feedback into a set of cost function parameters. The active elicitation interaction is driven by two heuristic query generation algorithms.

I have shown how ARNAULD allows SUPPLE to be adapted to a person's subjective preferences and illustrated this with examples of user interfaces generated with two different cost functions learned using ARNAULD.

In Chapter 5, I introduced the ABILITY MODELER which—after directing the user to perform a one-time set of diagnostic tasks—builds a personalized model of the user's *objective* motor abilities as applied to controlling the mouse pointer. This personalized model of the user's abilities is then used as an input to SUPPLE to automatically generate user interfaces that are predicted to be the fastest to use for that particular person. That chapter also introduced an extension to the SUPPLE system that allowed user interfaces to be easily adjusted to accommodate users' different vision abilities. Importantly, these two types of

adaptation—to motor and vision abilities—can be used together, allowing SUPPLE to adapt to people with a combination of motor and vision impairments, a population that is poorly served by current assistive technologies.

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 many fewer errors using automatically-generated personalized interfaces compared to the default user interfaces. Additionally, participants with motor impairments strongly preferred automatically generated user interfaces to the default ones. All these results were particularly strong and consistent for ability-based interfaces adapted to the participants' actual motor capabilities using models generated by the ABILITY MODELER: participants with motor impairments were between 8.4% and 42.2% (mean=26.4%) faster with those interfaces than with the manufacturers' defaults, they preferred those interfaces to all others, and they found those interfaces the easiest to use, the most efficient, and least physically tiring. 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 current difference in performance between users with motor impairments and able-bodied users is at least partially due to user interfaces being designed with a particular set of assumptions in mind—assumptions that are inaccurate for users with motor impairments. By generating personalized interfaces, which reflect these users' unique capabilities, I have shown that it is possible to greatly improve the speed and accuracy of users with motor impairments, even when they use standard input devices such as mice and trackballs.

These results also confirm that the right trade-off in user interface design depends on a particular user's individual capabilities. Even able-bodied participants were faster and made fewer errors with ability-based interfaces, and even they recognized these interfaces as significantly easier to use than the alternatives. In the end, however, they found the ability-based interfaces, which exchanged sparseness and familiar aesthetics for improved ease of manipulation, to be uglier and generally no more preferable than the baselines.

Finally, motivated by the controversy surrounding user interfaces that automatically adapt to user's behavior, as well as the relative lack of empirical evidence to inform the debate, I conducted three studies to explore different points in the design space of such adaptive user interfaces and presented the results in Chapter 6. Through a discussion of my own and past results, I have identified a number of properties of adaptive user interfaces that are likely to impact their acceptance by the users. In particular, this body of work suggests that Split Interfaces, which duplicate (rather than move) functionality to a clearly designated adaptive part of the interface, without modifying the main part of the interface, tend to improve users' performance and satisfaction compared to non-adaptive baselines.

Among other design dimensions, I have examined the influence of accuracy and predictability on adaptive toolbar user interfaces. The results of my particular study show that both predictability and accuracy affect participants' satisfaction but only accuracy had a significant effect on user performance or utilization of the adaptive interface. Contrary to my expectations, improvement in accuracy had a stronger effect on performance, utilization and some satisfaction ratings than the improvement in predictability. These results suggest that even though machine learning algorithms may produce inscrutable behavior, in certain cases they may have the potential to improve user satisfaction. Specifically, if a machine learning algorithm can more accurately predict a user's next action or a parameter value, then it may outperform a more predictable method of selecting adaptive buttons or default values. However, because predictability and accuracy affect different aspects of users' satisfaction, improvements to one of these factors cannot fully offset the losses to the other.

7.2 Future Work

An exciting next step will be to deploy the tools developed in this dissertation. A recently adopted V2 standard [6] for abstractly representing interface capabilities of applications and appliances was designed with the hope of allowing users a personalized remote control access to those applications and appliances. Limited prototypes of a V2-compliant Universal Remote Console have been developed [155, 154] but SUPPLE would provide a more complete solution.

An important limitation to the wider adoption of any model-based interface tools, including SUPPLE, is that they require designers to start by explicitly defining an abstract user interface model [98]. This is contrary to standard design practice, where designers typically start by exploring multiple low fidelity concrete interface prototypes and only then iteratively refine and formalize the details of the interaccction [80]. Therefore, an important future work direction is to create tools that can *infer* the functional specification by observing designers as they manually create the default user interfaces for the applications.

In my dissertation, I considered two metrics to optimize user interfaces for, namely, the models of users' preferences and 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 of and quality of knowledge contributions. Kohavi et al., [76] offer some helpful initial insights.

Another promising direction will be to pursue semantic adaptation of user interfaces. In contrast to my dissertation research, where I adapted the *structure and presentation* of the interfaces, future work could explore ways to automatically adapt the *functionality* itself; that is, the ways to automatically simplify user interfaces. This is an important problem because solving it would enable complex applications to be transformed for 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 [91]. This is a hard problem to solve automatically because it requires understanding of the function of interface elements. The existing solutions rely on extensive semantic annotations by the designer or by the user to perform such adaptations [35]. An alternative approach would be to leverage large user communities by automatically mining usage and customization traces.

There are also many more opportunities for further exploration of the design space of adaptive user interfaces. In particular, both my work and previous research indicate that the user acceptance of an adaptive interface is partially dependent on how much time and effort the user stands to save by using adaptation. This suggests that users with impairments are likely to benefit from different—perhaps more aggressive—adaptive approaches than able-bodied users. Yet, there have been no studies of adaptive interfaces involving users with impairments.

7.3 Parting Thoughts

SUPPLE is not intended to replace human designers. 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. Instead, SUPPLE offers alternative user interfaces for those users, whose devices, tasks, preferences, and abilities 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. My work demonstrates that automated tools are a feasible way of addressing this scalability challenge: my SUPPLE system can generate user interfaces in a matter of seconds and all the personalization mechanisms I developed rely entirely on user input and do not require any expert assistance.

BIBLIOGRAPHY

[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] M. Akamatsu, I. S. MacKenzie, and T. Hasbroucq. A comparison of tactile, auditory, and visual feedback in a pointing task using a mouse-type device. *Ergonomics*, 38(4):816–827, April 1995.

[6] ANSI/INCITS. Protocol to facilitate operation of information and electronic products through remote and alternative interfaces and intelligent agents, 2005. ANSI/INCITS 393-2005.

[7] Antoine Arnauld. *La logique, ou l'art de penser*. Chez Charles Savreux, au pied de la Tour de Nostre Dame, Paris, 1662.

[8] Greg J. Badros, Alan Borning, and J. Stuckey. The Cassowary Linear Arithmetic Constraint Solving Algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(4), 2001.

[9] David Benyon. Adaptive systems: A solution to usability problems. *User Modeling and User-Adapted Interaction*, 3(1):65–87, March 1993.

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

[11] Jeffrey P. Bigham, Ryan S. Kaminsky, Richard E. Ladner, Oscar M. Danielsson, and Gordon L. Hempton. Webinsight:: making web images accessible. In *Assets '06: Proceedings of the 8th international ACM SIGACCESS conference on Computers and accessibility*, pages 181–188, New York, NY, USA, 2006. ACM Press.

[12] 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.

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

[14] 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.

[15] Craig Boutilier. A POMDP formulation of preference elicitation problems. In *AAAI/IAAI*, pages 239–246, 2002.

[16] Craig Boutilier, Relu Patrascu, Pascal Poupart, and Dale Schuurmans. Constraint-based optimization with the minimax decision criterion. In *Ninth International Conference on Principles and Practice of Constraint Programming*, 2003.

[17] PE Bravo, M. LeGare, AM Cook, and S. Hussey. A study of the application of Fitts' law to selected cerebral palsied adults. *Percept Mot Skills*, 77(3 Pt 2):1107–17, 1993.

[18] PE Bravo, M. Legare, AM Cook, and SM Hussey. Application of Fitts law to arm movements aimed at targets in people with cerebral palsy. *Proceedings of the 13 thAnnual Conference of the Rehabilitation Engineering Society of North America*, 1990.

[19] Andrea Bunt, Cristina Conati, and Joanna McGrenere. What role can adaptive support play in an adaptive system. In *Proceedings of IUI'04*, Funchal, Portugal, 2004.

[20] Christopher J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Min. Knowl. Discov.*, 2(2):121–167, 1998.

[21] Robin D. Burke, Kristian J. Hammond, and Benjamin C. Young. The findme approach to assisted browsing. *IEEE Expert: Intelligent Systems and Their Applications*, 12(4):32–40, 1997.

[22] Scott Carter, Amy Hurst, Jennifer Mankoff, and Jack Li. Dynamically adapting GUIs to diverse input devices. In *Assets '06: Proceedings of the 8th international ACM SIGACCESS conference on Computers and accessibility*, pages 63–70, New York, NY, USA, 2006. ACM Press.

[23] Ben Carterette, Paul N. Bennett, David Maxwell Chickering, and Susan T. Dumais. Here or there: Preference judgments for relevance. In *Proceedings of the European Conference on Information Retrieval (ECIR)*, 2008. To appear.

[24] U. Chajewska, D. Koller, and D. Ormoneit. Learning an agent's utility function by observing behavior. In *Proceedings of ICML'01*, 2001.

[25] Urszula Chajewska and Daphne Koller. Utilities as random variables: Density estimation and structure discovery. In *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pages 63–71. Morgan Kaufmann Publishers Inc., 2000.

[26] Urszula Chajewska, Daphne Koller, and Ronald Parr. Making rational decisions using adaptive utility elicitation. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 363–369. AAAI Press / The MIT Press, 2000.

[27] E. Chicowski. It's all about access. *Alaska Airlines Magazine*, 28(12):26–31, 80–82, 2004.

[28] Andy Cockburn, Carl Gutwin, and Saul Greenberg. A predictive model of menu performance. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 627–636, New York, NY, USA, 2007. ACM.

[29] 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.

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

[31] 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.

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

[33] Leah Findlater and Joanna McGrenere. Impact of screen size on performance, awareness, and user satisfaction with adaptive graphical user interfaces. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 1247–1256, New York, NY, USA, 2008. ACM.

[34] 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.

[35] 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.

[36] 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.

[37] 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 Press.

[38] 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.

[39] 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.

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

[41] 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.

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] Krzysztof Z. Gajos and Daniel S. Weld. Usable AI: Experience and reflections. In *Workshop on Usable Artificial Intelligence (at CHI'08)*, 2008.

[48] Krzysztof Z. Gajos, Daniel S. Weld, and Jacob O. Wobbrock. Decision-theoretic user interface generation. In *AAAI'08*. AAAI Press, 2008.

[49] 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.

[50] Melinda T. Gervasio, Michael D. Moffitt, Martha E. Pollack, Joseph M. Taylor, and Tomas E. Uribe. Active preference learning for personalized calendar scheduling assistance. In *IUI '05: Proceedings of the 10th international conference on Intelligent user interfaces*, pages 90–97, New York, NY, USA, 2005. ACM Press.

[51] S. Greenberg and I.H. Witten. Adaptive personalized interfaces: A question of viability. *Behaviour & Information Technology*, 4(1):31–45, 1985.

[52] 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.

[53] A. Gump, M. LeGare, and D. L. Hunt. Application of fitts' law to individuals with cerebral palsy. *Percept Mot Skills*, 94(3 Pt 1):883–895, June 2002.

[54] Susumu Harada, James A. Landay, Jonathan Malkin, Xiao Li, and Jeff A. Bilmes. The vocal joystick:: evaluation of voice-based cursor control techniques. In *Assets '06: Proceedings of the 8th international ACM SIGACCESS conference on Computers and accessibility*, pages 197–204, New York, NY, USA, 2006. ACM Press.

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

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

[57] 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.

[58] T. Hastie, R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning*. Springer, August 2001.

[59] 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.

[60] D. Heckerman, E. Horvitz, and B. Middleton. An approximate nonmyopic computation for value of information. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(3):292–298, 1993.

[61] 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.

[62] 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.

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

[64] 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.

[65] Eric Horvitz. Principles of mixed-initiative user interfaces. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 159–166, New York, NY, USA, 1999. ACM Press.

[66] Eric Horvitz and Johnson Apacible. Learning and reasoning about interruption. In *Proceedings of the 5th international conference on Multimodal interfaces*, pages 20–27. ACM Press, 2003.

[67] Eric Horvitz, Andy Jacobs, and David Hovel. Attention-sensitive alerting. In *Proceedings of the 15th Annual Conference on Uncertainty in Artificial Intelligence (UAI-99)*, pages 305–313, San Francisco, CA, 1999. Morgan Kaufmann Publishers.

[68] Eric Horvitz, Paul Koch, and Johnson Apacible. Busybody: creating and fielding personalized models of the cost of interruption. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 507–510, New York, NY, USA, 2004. ACM Press.

[69] 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.

[70] International Organization for Standardization. 9241-9 Ergonomic requirements for office work with visual display terminals (VDTs)-Part 9: Requirements for non-keyboard input devices, 2000.

[71] 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.

[72] S. Keates, P.J. Clarkson, L.A. Harrison, and P. Robinson. *Towards a practical inclusive design approach*. ACM Press New York, NY, USA, 2000.

[73] 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.

[74] Ralph L. Keeney and Howard Raiffa. *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. John Wiley and Sons, 1976. Republished in 1993 by Cambridge University Press.

[75] 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.

[76] 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.

[77] S. Kurniawan, A. King, D.G. Evans, and P. Blenkhorn. Design and user evaluation of a joystick-operated full-screen magnifier. *Proceedings of the conference on Human factors in computing systems*, pages 25–32, 2003.

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

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

[80] 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.

[81] Greg Linden, Steve Hanks, and Neal Lesh. Interactive assessment of user preference models: The automated travel assistant. In *Proceedings, User Modeling '97*, 1997.

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

[83] Edmund Lopresti, David M. Brienza, Jennifer Angelo, Lars Gilbertson, and Jonathan Sakai. Neck range of motion and use of computer head controls. In *Assets '00: Proceedings of the fourth international ACM conference on Assistive technologies*, pages 121–128, New York, NY, USA, 2000. ACM Press.

[84] R.L. Mace, G.J. Hardie, P. Jaine, and North Carolina State University Center for Universal Design. *Accessible Environments: Toward Universal Design*. Center for Accessible Housing, North Carolina State University, 1990.

[85] 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.

[86] I. S. Mackenzie. Fitts' law as a research and design tool in human-computer interaction. *Human-Computer Interaction*, 7(1):91–139, 1992.

[87] 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.

[88] Pattie Maes. Agents that reduce work and information overload. *C. ACM*, 37(7):31–40, 146, 1994.

[89] Jennifer Mankoff, Anind Dey, Udit Batra, and Melody Moore. Web accessibility for low bandwidth input. In *Assets '02: Proceedings of the fifth international ACM conference on Assistive technologies*, pages 17–24, New York, NY, USA, 2002. ACM Press.

[90] L. McGinty and B. Smyth. Tweaking Critiquing. *Proceedings of the Workshop on Personalization and Web Techniques at the International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 20–27, 2003.

[91] 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.

[92] 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.

[93] 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.

[94] Bradley N. Miller, Joseph A. Konstan, and John Riedl. Pocketlens: Toward a personal recommender system. *ACM Trans. Inf. Syst.*, 22(3):437–476, 2004.

[95] Thomas P. Minka. Expectation propagation for approximate bayesian inference. In *UAI '01: Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence*, pages 362–369, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[96] J. Mitchell and B. Shneiderman. Dynamic versus static menus: an exploratory comparison. *SIGCHI Bull.*, 20(4):33–37, April 1989.

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

[98] 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.

[99] Brad A. Myers, Rishi Bhatnagar, Jeffrey Nichols, Choon H. Peck, Dave Kong, Robert Miller, and Chris A. Long. Interacting at a distance: measuring the performance of laser pointers and other devices. In *CHI '02: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 33–40, New York, NY, USA, 2002. ACM Press.

[100] J. Nichols, B.A. Myers, M. Higgins, J. Hughes, T.K. Harris, R. Rosenfeld, and M. Pignol. Generating remote control interfaces for complex appliances. In *Proceedings of UIST'02*, Paris, France, 2002.

[101] 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.

[102] 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.

[103] 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.

[104] Jakob Nielsen. Enhancing the explanatory power of usability heuristics. In *CHI '94: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 152–158, New York, NY, USA, 1994. ACM Press.

[105] 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.

[106] 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.

[107] 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.

[108] 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.

[109] 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.

[110] Fabio Paterno, Cristiano Mancini, and Silvia Meniconi. Concurtasktrees: A diagrammatic notation for specifying task models. In *INTERACT*, pages 362–369, 1997.

[111] 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.

[112] Tim F. Paymans, Jasper Lindenberg, and Mark Neerincx. Usability trade-offs for adaptive user interfaces: ease of use and learnability. In *IUI '04: Proceedings of the 9th international conference on Intelligent user interface*, pages 301–303, New York, NY, USA, 2004. ACM Press.

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

[114] 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.

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

[116] P. Pu, B. Faltings, and M. Torrens. User-involved preference elicitation. In *IJCAI'03 Workshop on Configuration*, Acapulco, Mexico, 2003.

[117] AR Puerta. A model-based interface development environment. *Software, IEEE*, 14(4):40–47, 1997.

[118] RS Rao, R. Seliktar, and T. Rahman. Evaluation of an isometric and a position joystick in a targetacquisition task for individuals with cerebral palsy. *Rehabilitation Engineering, IEEE Transactions on [see also IEEE Trans. on Neural Systems and Rehabilitation]*, 8(1):118–125, 2000.

[119] 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.

[120] 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.

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

[122] Anthony Savidis. Dynamic software assembly for automatic deployment-oriented adaptation. *Electronic Notes in Theoretical Computer Science*, 127(3):207–217, April 2005.

[123] Anthony Savidis, Margherita Antona, and Constantine Stephanidis. A decision-making specification language for verifiable user-interface adaptation logic. *International Journal of Software Engineering And Knowledge Engineering*, 15(6):1063 – 1094, 2005.

[124] Anthony Savidis and Constantine Stephanidis. Inclusive development: Software engineering requirements for universally accessible interactions. *Interacting with Computers*, 18(1):71–116, January 2006.

[125] Greg Schohn and David Cohn. Less is more: Active learning with support vector machines. In *ICML*, pages 839–846, 2000.

[126] 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.

[127] Ingrid U. Scott, William J. Feuer, and Julie A. Jacko. Impact of graphical user interface screen features on computer task accuracy and speed in a cohort of patients with age-related macular degeneration. *American Journal of Ophthalmology*, 134(6):857–862, December 2002.

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

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

[130] Juliet P. Shaffer. Multiple hypothesis-testing. *ANNUAL REVIEW OF PSYCHOLOGY*, 46:561–584, 1995.

[131] Sybil Shearin and Henry Lieberman. Intelligent profiling by example. In *IUI '01: Proceedings of the 6th international conference on Intelligent user interfaces*, pages 145–151. ACM Press, 2001.

[132] B. Shneiderman and P. Maes. Direct manipulation vs. interface agents. *Interactions*, 4(6):42–61, 1997.

[133] Smits-Engelsman, B., Rameckers, E., Duysens, and J. Children with congenital spastic hemiplegia obey fitts law in a visually guided tapping task. *Experimental Brain Research*, 177(4):431–439, March 2007.

[134] C. Stephanidis. Towards the Next Generation of UIST: Developing for all Users. *Proceedings of the Seventh International Conference on Human-Computer Interaction-Volume 1-Volume I table of contents*, pages 473–476, 1997.

[135] C. Stephanidis. User interfaces for all: New perspectives into human-computer interaction. In C. Stephanidis, editor, *User Interfaces for All*, pages 3–17. Lawrence Erlbaum, 2001.

[136] 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.

[137] 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.

[138] 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.

[139] Simon Tong and Daphne Koller. Support vector machine active learning with applications to text classification. *Journal of Machine Learning Research*, 2:45–66, 2001.

[140] Robert Trevellyan and Dermot P. Browne. A self-regulating adaptive system. In *CHI '87: Proceedings of the SIGCHI/GI conference on Human factors in computing systems and graphics interface*, pages 103–107, New York, NY, USA, 1987. ACM Press.

[141] Theophanis Tsandilas and Schraefel. An empirical assessment of adaptation techniques. In *CHI '05: CHI '05 extended abstracts on Human factors in computing systems*, pages 2009–2012, New York, NY, USA, 2005. ACM Press.

[142] 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.

[143] 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.

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

[145] 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.

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

[147] 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.

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

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

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

[151] 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.

[152] 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.

[153] 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.

[154] G. Zimmermann, G. Vanderheiden, M. Ma, M. Gandy, S. Trewin, S. Laskowski, and M. Walker. Universal remote console standard: toward natural user interaction in ambient intelligence. *Conference on Human Factors in Computing Systems*, pages 1608–1609, 2004.

[155] Gottfried Zimmermann, Gregg Vanderheiden, and Al Gilman. Prototype implementations for a universal remote console specification. In *CHI '02 extended abstracts on Human factors in computer systems*, pages 510–511, Minneapolis, Minnesota, USA, 2002. ACM Press.

VITA

Krzysztof Gajos received his B.Sc. and M.Eng. degrees in Computer Science from MIT in 1999 and 2000, respectively. For the next two years, he was a research scientist at the MIT Artificial Intelligence Laboratory, where he managed The Intelligent Room Project. From 2002, he attended the University of Washington, where he worked with Daniel Weld and Jacob Wobbrock. He was a recipient of a Microsoft Graduate Research Fellowship and he has also been a visiting faculty member at the Ashesi University in Ghana where he designed and taught an introductory course in artificial intelligence. In 2003, he received M.Sc. and in 2008 a Ph.D., both in Computer Science at the University of Washington.