In Proceedings of Ubicomp 2005, Tokyo, Japan

Fast And Robust Interface Generation for Ubiquitous Applications

Krzysztof Gajos, David Christianson, Raphael Hoffmann, Tal Shaked, Kiera Henning, Jing Jing Long, and Daniel S. Weld

University of Washington
Seattle, WA, USA
{kgajos,dbc1,raphelh,tshaked,kierah,jingjing,weld}@cs.washington.edu
http://www.cs.washington.edu/ai/supple/

Abstract. We present Supple, a novel toolkit which automatically generates interfaces for ubiquitous applications. Designers need only specify declarative models of the interface and desired hardware device and Supple uses decision-theoretic optimization to automatically generate a concrete rendering for that device. This paper provides an overview of our system and describes key extensions that barred the previous version (reported in [3]) from practical application. Specifically, we describe a functional modeling language capable of representing complex applications. We propose a new adaptation strategy, *split interfaces*, which speeds access to common interface features without disorienting the user. We present a *customization facility* that allows designers and end users to override Supple's automatic rendering decisions. We describe a distributed architecture which enables computationally-impoverished devices to benefit from Supple interfaces. Finally, we present experiments and a preliminary user-study that demonstrate the practicality of our approach.

1 Introduction

The growth of mobile and ubiquitous computing has caused increased dependance on digitally encoded information and has increased users' expectations of being able to access, create and manipulate digital content in a variety of situations. As a result users frequently rely on devices other than desktop computers for their digital needs. These devices include mobile phones, PDAs, tablet computers, touch panels, and increasingly wall-sized displays operated by finger or laser pointing. These devices not only differ in their screen size and resolution; they also support very diverse kinds of input devices and modes of interaction.

These trends put an enormous pressure on software developers to make their products available for a large number of platforms. While the logic of the application may often transfer easily across different platforms, the user interfaces typically have to be designed and implemented from scratch. Of course, interface design is always time-consuming, but there are several aspects of ubiquitous applications that are especially challenging: 1) New kinds of devices enter the market at a rapid pace. 2) Since ubiquitous computing is a young field, there is a smaller track record of successful interface designs and increased need for iterative prototyping. 3) In ubiquitous computing new functionality often emerges from interactions of spontaneously aggregated devices and services [10].

This paper describes SUPPLE, a fast and efficient UI toolkit for ubiquitous applications which addresses these issues by automatically generating a personalized interface for a wide range of hardware platforms. We argue that such a toolkit should satisfy the five requirements listed below.

– **Easy:** The toolkit should support rapid prototyping and be easy for application developers to use. Section 2 explains how SUPPLE's high-level interface representation speeds development, and Table 2 in Section 6 details the code complexity of the examples in this paper.
– **Capable:** The toolkit should handle complex interfaces and rich data types. Section 2 describes our expanded modeling language, and Table 3 demonstrates that SUPPLE can render reasonably complex interfaces very quickly.
– **Adaptive:** Generated interfaces should possess adaptation and customization features to make convenient operation by users with widely varying activities, styles and preferences. However, automated changes to an interface must minimize the chance of user disorientation. Section 3 introduces a novel interface-adaptation mechanism called *split interfaces* and describes a preliminary user study suggesting user acceptance. Section 4 presents a complementary customization capability, which allows users to tailor any SUPPLE interface to their desires as well as to undo unwanted adaptations.
– **Portable:** A single interface specification should enable rendering that interface on every supported hardware platform, including devices which are computationally impoverished and which may not have network connectivity. Section 2 explains how SUPPLE takes a device description as input and uses a decision-theoretic optimization algorithm to render an interface specifically tailored to the capabilities and constraints of that hardware platform. Section 5 describes our distributed architecture and caching infrastructure, which supports remote rendering and wireless operation. Section 6 provides preliminary performance data for our system.
– **Extensible:** It should be easy to extend the set of hardware devices supported by the toolkit. The most difficult aspect of getting SUPPLE to generate interfaces for a new hardware platform is specifying the cost model [3] for that device's interaction modes. In the past, this was indeed a laborious process, but we have recently developed a preference-elicitation methodology and machine-learning algorithm which quickly generates these cost models. Space precludes a description of our technique, but see [4] for details and experiments showing its efficacy.

The next section summarizes SUPPLE's rendering algorithm and describes its extended modeling language which now supports more complex applications and data types required by ubiquitous applications. Section 3 explains SUPPLE's novel method of supporting dynamic improvements to the UI in a manner which doesn't disorient the user; a preliminary user-study validates the approach. Section 4 presents a mechanism to allow a designer to override decisions made by

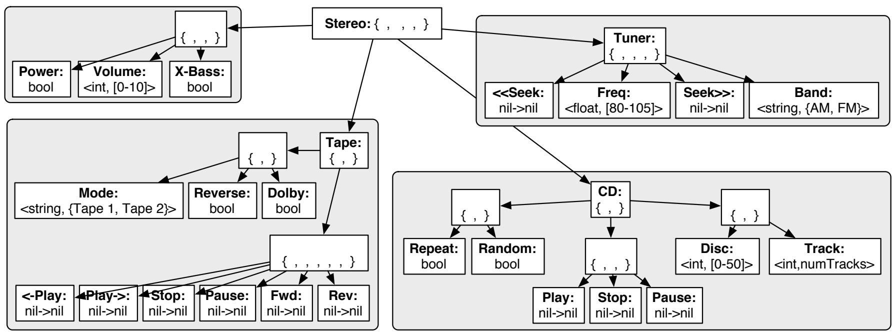


Fig. 1. Graphical representation of the functional specification for a stereo controller. For clarity, different parts of the specification are grouped with gray shading.

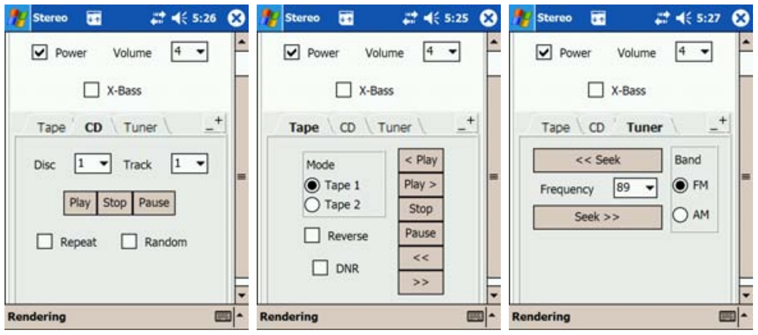


Fig. 2. Three tab views of the stereo specification, rendered for a PDA.

the automatic rendering engine. Section 5 describes the distributed architecture, which enables Supple to run on computationally impoverished platforms such as PDAs. Section 6 presents experimental results, which confirm Supple's fitness as a toolkit for ubiquitous applications. We end the paper with a discussion of related and future research.

2 Overview of Supple

Supple takes three inputs: a *functional specification* of the interface, a *device model* and a *user model*. The functional specification defines the *types* of data that need to be exchanged between the user and the application (*e.g.* Figure 1). The device model describes which widgets are available on the device and provides a *cost function*, which estimates the user effort required to manipulate these widgets with the interaction methods supported by the device. Finally, a user's typical activities are modeled with a device- and rendering-independent *user trace*. Supple's rendering algorithm combines constraint propagation with branch-and-bound search, guided by an admissible heuristic.

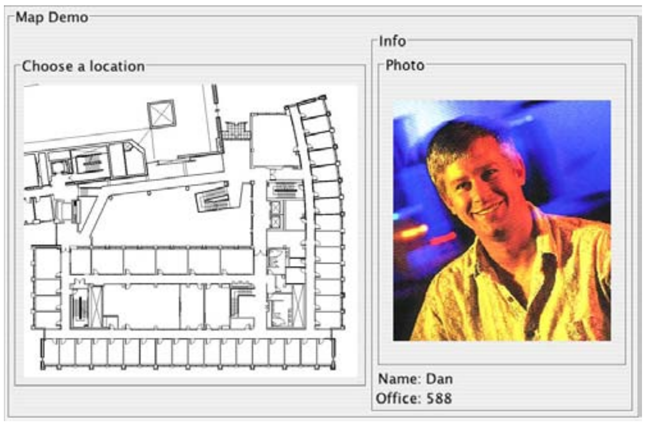


Fig. 3. An interface utilizing images and clickable maps.

In this section, we describe SUPPLE's functional specification language, briefly discuss its optimization-based rendering algorithm and provide an overview of its implementation, focusing on aspects not elucidated in [3].

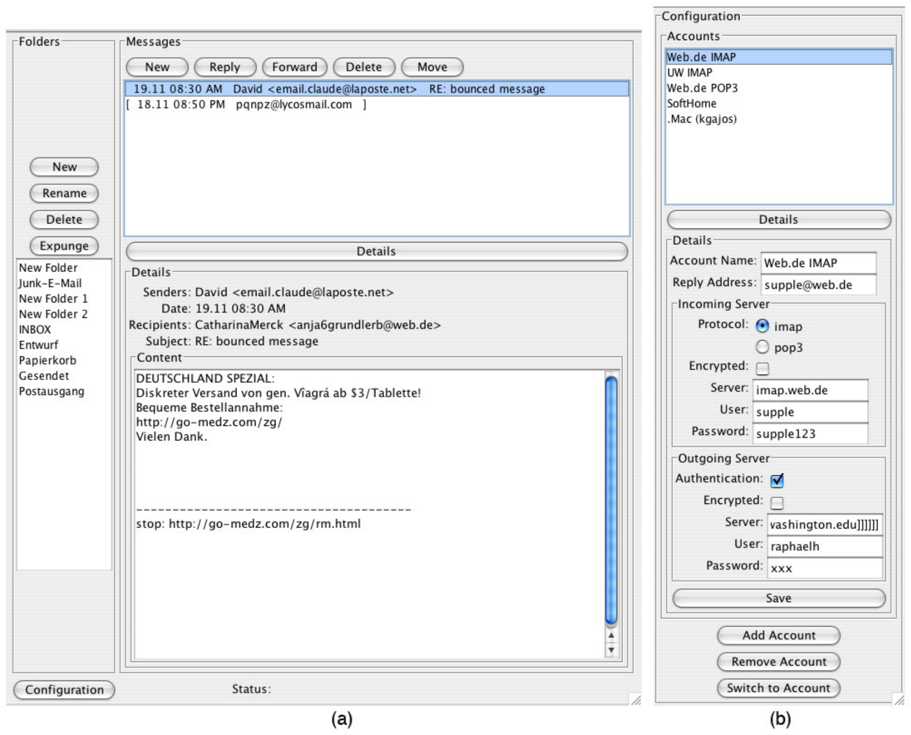
2.1 Functional Specification Language for Interface Modeling

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. Each element is defined in terms of its 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 switch for the stereo system is represented as a Boolean in the specification of Figure 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 shown in Figure 3, where a user can point at different offices on a building map, causing the occupant's image to be displayed.

Container types, formally represented as $\{\tau_1, \tau_2, \ldots, \tau_n\}$, are used to create groups (or records) of simpler elements. For example, all of the interior nodes (*e.g.*, Tuner, Tape, CD, Stereo or the unnamed intermediate nodes) in the specification tree in Figure 1 are instances of the container type.

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 stereo example, the volume is defined as an integer type whose values are constrained to lie between 0 and 10. In the email client shown in Figure 4(a) 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 (note here that the constraints on the legal values of an element can change dynamically at run time *e.g.*, when new folders are created). In most



19.11 08:30 AM David <email.claude@laposte.net> RE: bounced message
[18.11 08:50 PM pqnpz@lycosmail.com]

New Folder
Junk-E-Mail
New Folder 1
New Folder 2
INBOX
Entwurf
Papierkorb
Gesendet
Postausgang

Details
Senders: David <email.claude@laposte.net>
Date: 19.11 08:30 AM
Recipients: CatharinaMerck <anja6grundlerb@web.de>
Subject: RE: bounced message

Content
DEUTSCHLAND SPEZIAL:
Diskreter Versand von gen. Vìagrá ab $3/Tablette!
Bequeme Bestellannahme:
http://go-medz.com/zg/
Vielen Dank.

----
stop: http://go-medz.com/zg/rm.html

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

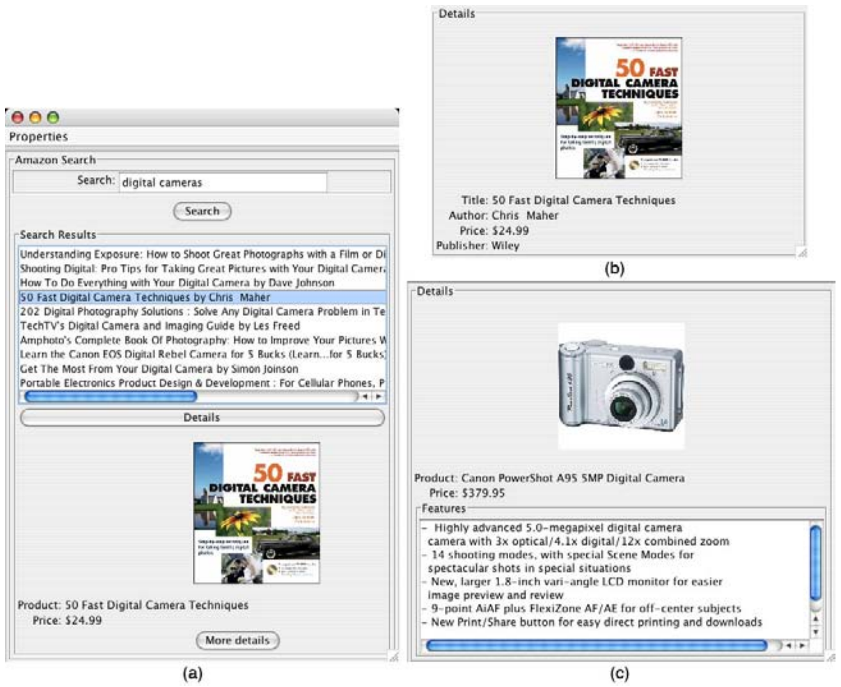
Details
Account Name: Web.de IMAP
Reply Address: supple@web.de
Incoming Server
Protocol: [✓ imap] pop3
Encrypted: ☐
Server: imap.web.de
User: supple
Password: supple123
Outgoing Server
Authentication: ✓
Encrypted: ☐
Server: vashington.edu]]]]]
User: raphaelh
Password: xxx

Fig. 4. An email client that uses Supple to render its user interface. (a) The main view. (b) The configuration pane.

cases elements of the constrained type are rendered as a discrete selection widget (list, combo box, etc) except for number ranges where continuous selectors such as sliders may be used.

Constraints can also be specified for container types. For example, consider the list of available email accounts in the email example of Figure 4(b). 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 she also wants to select different accounts to view. Thus, we model the interface element representing the current account as a container object whose domain of values is restricted to all 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 4(b). 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.

While this approach makes modeling easy, it assumes that all the objects in a container's domain are of exactly the same type. In practice, this is not



Search: digital cameras

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

Product: 50 Fast Digital Camera Techniques
Price: $24.99

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

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

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

Fig. 5. 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 more specialized view for each item. (b) Detailed view for a book. (c) Detailed view for a digital camera.

always the case. For example, consider Figure 5's interface to Amazon Web Services. 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 5(a)). Any time a specialized object is selected by the user, another button is highlighted, alerting the user that more detailed information is available, which can be displayed in a separate window as in Figures 5(b) and (c).

Vectors: elements of type $vector(\langle \tau, \mathcal{C}_\tau \rangle)$ are used to support multiple selection; they denote an ordered sequence of zero or more values of type τ. The constraints $\mathcal{C}_\tau$ define the set of values of type τ that can be selected from. For example, the list of emails in the email client (Figure 4(a)) is represented as a

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

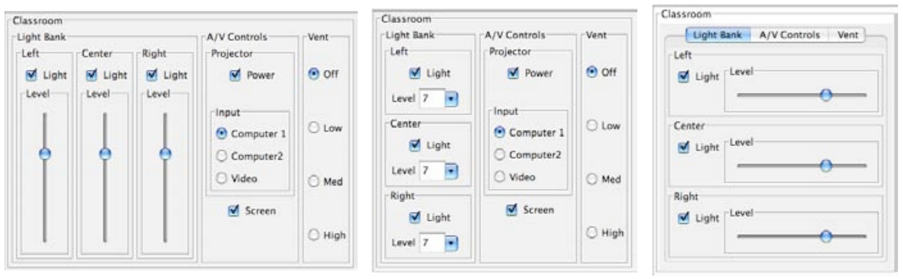


Fig. 6. SUPPLE optimally uses the available space and robustly degrades the quality of the rendered interface if presented with a device with a smaller screen size. This figure shows three renderings of a classroom controller on three devices with progressively narrower screens.

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 *i.e.*, the interface component that is to be displayed after the typical execution of the action. Unlike the other types which are used to represent the application's *state*, the action type is used to invoke the *methods*. The "Reply" button in the rendering of the email client interface in Figure 4(a) is represented as an action with a null parameter type (since it operates on the current message) and Message as the return type. The Search in the Amazon browser example in Figure 5(a) is an action with String as a parameter type and a null return type.

2.2 Algorithm

Unlike previous model-based rendering systems (*e.g.* the Personal Universal Controller [6]), which use templates or rule-based approaches to generating user interfaces, SUPPLE uses decision-theoretic, combinatorial optimization. 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 and a novel, admissible heuristic to explore the space of candidate renderings; full constraint propagation is used to maximize the pruning effect of violated constraints. Several variations on the algorithm are empirically evaluated in [3].

SUPPLE's use of a cost function to guide the choice of rendering has several advantages. Unlike a rule set that needs to be created by hand, the cost function can be quickly constructed automatically from designer's responses to examples of concrete renderings of different interfaces [4]. Furthermore, optimization is relatively robust, flexibly handling tradeoffs and interactions between choices in different parts of the interface. In contrast, rule-based systems are fragile because small changes to device constraints (*e.g.*, display size) may require a

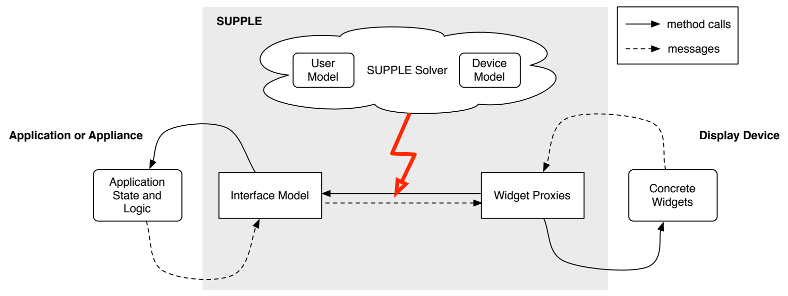


Fig. 7. Supple's implementation: The interface model exposes the state variables and methods that should become accessible through the interface. The widget proxies generated by the device model are assigned to interface model elements by Supple's optimization algorithm.

major change to the rendering. A rule derivation which works well on one PDA might fail when used with a slightly smaller screen, since the candidate interface might no longer fit. Conversely, 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 6 illustrates how Supple robustly degrades the quality of the generated user interfaces as it is presented with devices with progressively narrower screens.

2.3 Implementation

Supple is written in Java and is designed to integrate easily with existing application code — especially with applications whose state is maintained in 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 7. 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:

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 necessary, 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 (or equivalent elements on other platforms) need to be displayed at run-time. Supple renders them dynamically as needed.

3 Automatic Adaptation without Disorientation

Automatically generating interfaces on the fly opens up the possibility of incorporating knowledge about the user and his tasks into the design of the concrete interface. Our initial paper on Supple explained how analysis of a user trace enabled the system to automatically adapt an interface to a given user's work habits by fully replacing one rendering with another [3]. This approach, however, could result in dramatic changes to the interface and consequently disorient the user. In this section, we compare alternative designs for adaptive interfaces in an exploratory user study; we then describe our implementation of *split interfaces*, a technique which reconciles adaptivity with predictability.

3.1 Exploratory User Study

For our study, we compared four interfaces: two traditional and two adaptive siblings. One of the traditional interfaces organized operations hierarchically, and the other used a "flat" structure, in which operations are always visible. Because these structures have different strengths and weaknesses, we considered different adaptation methods in each case.

A hierarchical interface offers clear organization, but makes certain operations harder to access, so we employ adaptation in an effort to speed access to commonly used functionality. Following [13, 16, 2] we advocate a generalization of *split menus* which we call *split interfaces* (Figure 8(b)). We partition an interface into *static* and *dynamic* sections. The static portion always maintains the same organization, while the dynamic portion displays speedy ways to invoke commonly used functions that might be otherwise hard to access. Note that in contrast to Sears and Shneiderman [13], who removed recently used items from their original locations when ordering them near the head, we propose *duplicating* functionality when placing it in the dynamic portion of the interface.

A flat interface organization makes every operation ready-at-hand, but may promote so many operations, that the result confuses the user. Thus, we use adaptation in an effort to facilitate navigation, directing the user to commonly used options. Specifically, we develop the notion of *altered prominence*, dynamically highlighting commonly used keys (Figure 8(d)).

Our hypothesis was that the split interface would be more universally accepted, because the original (static) portion of the interface is never changed, maintaining a sense of user control. We expected altered prominence to be more controversial, because adaption modifies the interface in ways that users can't ignore.

For our experimental domain, we chose a graphing calculator application, because most tasks require numerous interactions with clickable UI elements. This provides a copious stream of user-activity data, allowing for relatively fast adaptation. We enrolled 16 users from a population of graduate students and staff in the Department of Computer Science & Engineering at the University of Washington. Half the users were assigned to the hierarchical/split interface and half to flat/altered prominence. We asked each user to complete two sets of 18 formula entry tasks. Users entered one set of formulae in an adaptive interface

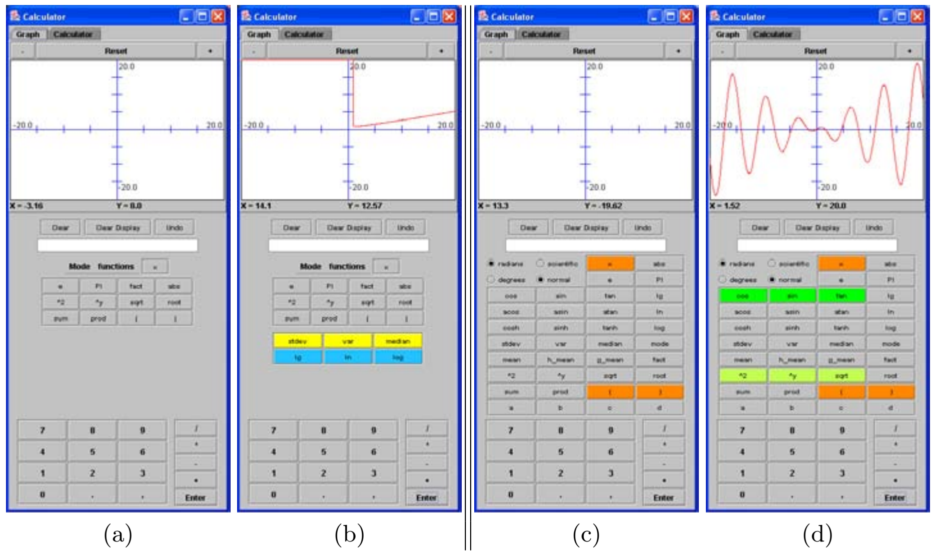
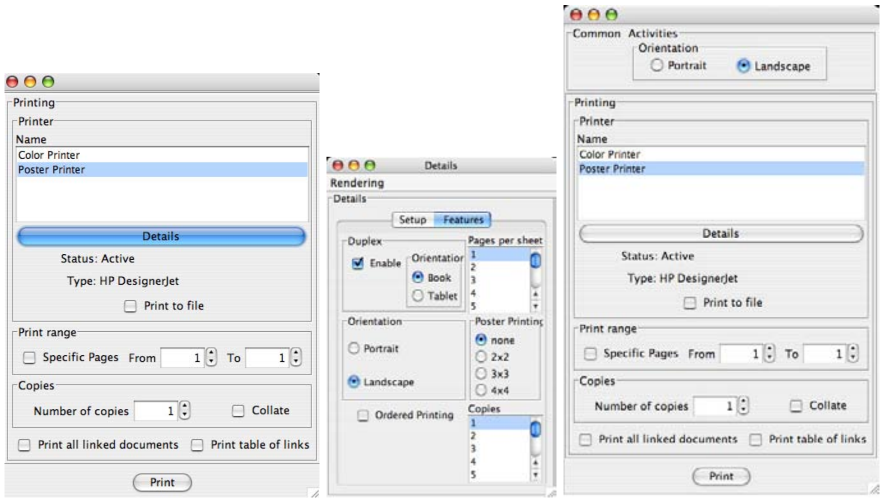


Fig. 8. The (handcrafted) interfaces used in our preliminary user study: (a) a static interface that requires users to use hierarchical pull-down menus to access advanced functions, (b) *split interface* — an adaptive version of the hierarchical interface in which families of the most frequently accessed functions are added to the (previously empty) "dynamic" area in the middle of the interface, (c) a static, flat interface with every button present at all times (d) *altered prominence* — an adaptive version of the flat interface, which highlights the two most recently used families of buttons. (The parenthesis and the x variable are highlighted at all times.)

and the other in a static UI. To neutralize learning effects, half the users started with the adaptive interface, while the other half first used the static UI. Users reported their subjective experience on two questionnaires.

Due to the moderate sample size, most of our results were not statistically significant. On average, the subjects completed the tasks faster using the adaptive interfaces. The speedup was very small for the split interface, but noticeable for the interface with altered prominence. Furthermore, when compared to the corresponding static interface, the error rate increased for the split interface, but decreased slightly for altered prominence.

The subjective comments were more surprising. Users considered altered prominence to be intuitive, but deemed the split interface to be slightly less intuitive than its static alternative. Interestingly, however, users expressed a strong and nearly-universal preference for the split interface when compared with its static version (this result was statistically significant with $p < 0.02$). In contrast, there was no statistically significant preference for altered prominence in the flat interface; in fact, some users expressed a strong dislike for that method of adaptation claiming that it caused them to get disoriented every time the highlighting changed.



(a) (b) (c)

Fig. 9. In the original print dialog box (a) it takes four mouse clicks to select landscape printing: details button, features tab (b), 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.

In both cases, the interface adapted at a rapid pace: several times during the course of a 10–15 minute-long session. We believe that both adaptation strategies would have had more impact if they were performed at a slower pace, giving the user a chance to notice and take full advantage of them. In addition, the prominence might better have been altered more gradually, so that the changes were imperceptible and hence not a distraction.

3.2 Adaptation in SUPPLE

Since our study indicated that users strongly favored split interfaces, we adopted this method in SUPPLE. Figure 9 shows an example where, in response to a particular user trace that repeatedly used "Landscape" printing, SUPPLE has automatically populated the "Common Activities" area of the top-level print dialog box with a one-click shortcut to this orientation. SUPPLE's split interfaces generally work well on interfaces that involve navigating among different windows or tab panes (or cards/pages on cell phones and pages on the Web).

Rendering a split interface is computationally harder than rendering an interface where each piece of functionality exists in exactly one place. In addition to finding the best assignment of widgets to interface elements, SUPPLE now must also decide what amount of space to set aside for the dynamic content, what functionality to represent in the dynamic area, and how to render it (duplicated functionality need not be rendered exactly as it was in the interface's static part).

Ideally, SUPPLE should duplicate functionality with the highest expected *utility* (*i.e.*, favoring the product of access likelihood and the difficulty of navigation with the static interface). Note, however, that the presence of a dynamic shortcut for an element E will likely reduce the chance that a user will navigate to E through the static interface. This in turn may cause a different rendering to be preferable for the static part. Thus the choice of static and dynamic aspects interact, and choosing the optimal static interface requires considering the *distribution* over possible dynamic content. Since this computation is intractable, we use a simple approximation — first choosing a static rendering in isolation, then greedily adding the best dynamic shortcuts in turn. We define the utility of a particular set of *duplicatedFunctionality* for a particular *concreteInterface* and a usage *trace* as:

$$\begin{aligned}&\texttt{utility}(\mathit{duplicatedFunctionality} \mid \mathit{concreteInterface}, \mathit{trace}) = \\ &\quad \texttt{expectedEffort}(\mathit{concreteInterface} \mid \mathit{trace}) \\ &\quad - \texttt{expectedEffort}(\mathit{concreteInterface}{+}\mathit{duplicatedFunctionality} \mid \mathit{trace})\end{aligned}$$

To compute `expectedEffort`, SUPPLE simulates use of the interface by replaying the trace of past interactions through a hypothesized interface and estimating the total "effort" required. The usage traces are recorded in terms of the functionality accessed and are stored in a rendering-independent manner. For desktop interfaces, the number of mouse clicks (for changing tabs, opening new windows, dismissing them, etc) is used as the approximation of the total effort. On WAP cell phones, the number of button presses is used as the metric. In cases where the functionality that the user wants to access has been duplicated in several parts of the interface, SUPPLE assumes that the user will use the most convenient copy of the functionality. While this model of expected effort is imperfect, our initial experience suggests that it is a good approximation for evaluating the potential utility of duplicated functionality.

In theory, a system might calculate the optimal percentage of space to allocate to the dynamic and static parts of an interface, respectively. Unfortunately, a principled approach to this optimization is computationally prohibitive. As a result, SUPPLE ignores this decision and only includes area for dynamic content if a user explicitly requests it or if the best available rendering does not fill all available space. Table 1 summarizes the algorithm that SUPPLE uses for selecting which interface elements to display in the "Common Activities" area of each interface view:[2]

4 Overriding Automated Rendering Decisions

SUPPLE's committment to completely-autonomous rendering hinders acceptance with users as it originally provided no means for either designers or the end users to control the presentation or organization of the final concrete interfaces. In

[2] A *view* is a unit of a user interface such as a window on a desktop computer or a card in a WML document.

```
selectCommonActivities(interfaceModel, concreteInterface, trace)
1.  initialize duplicatedFunctionality
2.  while there is still a view with space for duplicated functionality
3.    currentBestUtility ← 0
4.    currentBestCandidate ← null
5.    foreach view in concreteInterface that has space for duplicated functionality
6.      foreach element in interfaceModel
7.        temp ← duplicatedFunctionality + duplicate element into view
8.        if utility(temp | concreteInterface, trace) > currentBestUtility
9.          then currentBestCandidate ← duplicate element into view
10.   if (currentBestCandidate != null)
11.     then duplicatedFunctionality ← duplicatedFunctionality + currentBestCandidate
12.     else return duplicatedFunctionality
13. return duplicatedFunctionality
```

Table 1. SUPPLE's algorithm for selecting elements to be displayed in the "Common Activities" area of each interface view.

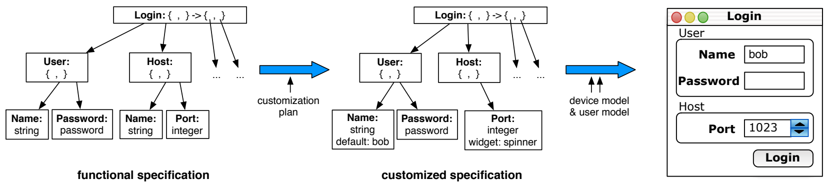


Fig. 10. SUPPLE's customization architecture. The user's customization actions are recorded in a *customization plan*. The next time the interface is rendered (possibly in a differently sized window or on a different device) the plan is used to transform the functional specification into a *customized specification* which is then rendered using decision-theoretic optimization as before. The interface shown in this figure is for a small FTP client.

order to encourage adoption (and satisfy requirements of ease and adaptivity), we devised a convenient way to override optimization-based choices. 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 are revealed. Duplication and rearrangement are specified with drag and drop.

Combining explicit customization with optimization-based rendering requires a major change to the original SUPPLE architecture (Figure 10). SUPPLE records a complete history of the user's customization operations in a *customization plan*. Rendering an interface proceeds in two phases. First, SUPPLE applies the plan to the functional specification in order to create a graph called a *customized* *specification*. In the second phase, SUPPLE applies decision-theoretic optimization (described in Section 2) to the device model, user model, and customized specification to render the interface.

The functional and customized specifications may have very different structures, *e.g.*, the customization specification may omit, duplicate or move functionality; it may also contain constraints on the set of widgets which may be used to render certain elements.

Note that the customization plan can be applied to *any* functional specification, including ones that the user (and SUPPLE) have not yet seen; it also may be applied when rendering an interface on a novel device (however, constraints requiring specific widgets may not be satisfiable). Explicitly representing the customization plan also allows SUPPLE to support a flexible undo system which encourages users to experiment with alternative interfaces.

5 Handling Computationally-Impoverished Devices

As required by the portability and extensibility requirements, SUPPLE supports the major modes of operation required by ubiquitous computing. In some situations, users will want to access applications on their desktop computers (PCs). It is also likely that users will want to use their desktop computers to access applications running 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 cellphones. These scenarios require that SUPPLE be able to present interfaces on devices other than those executing application logic.

Furthermore, our 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. Thus we require that SUPPLE be able to utilize a remote rendering service (we refer to it as the *solver server*) to accelerate the rendering process, whenever network connectivity is available. Figure 11 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 we summarize SUPPLE's distributed architecture.

Being able to render interfaces for remote application is a common feature of today's interface generation systems, like the previously mentioned Personal Universal Controller [5] and the Ubiquitous Interactor [7]. However, the computational complexity of our decision-theoretic rendering algorithm create unique challenges for the distribution of our system.

SUPPLE naturally supports distribution of the application and the interface, when its XML-based syntax is used to describe the functional specification. We have also implemented a distributed framework, based on Java RMI, that achieves the same result when the programatic interface is used. SUPPLE automatically choses local or remote bindings depending on the configuration, and the application programmer need not be aware that distributed operation is involved.

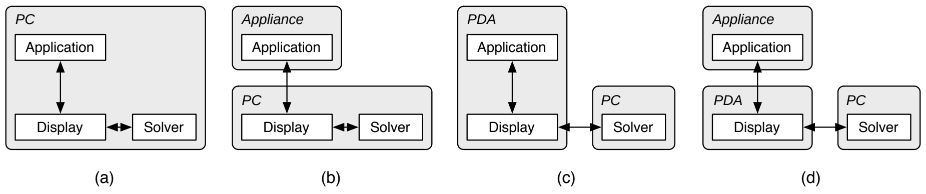


Fig. 11. SUPPLE allows the application, solver server, and interface to run on different devices; the following modes of operation are common: (a) An application running on a PC is displayed on the same machine — the user interface is rendered locally. (b) A remote application is displayed on a PC — 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.

Currently the remote solver server can only be invoked using Java RMI, because our device model is not fully declarative.[3] In the future, however, we plan to implement the solver server as a web service.

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

6 Quantitative Evaluation

In this Section, we evaluate SUPPLE's versatility, quantify the complexity of interface specifications, report on the rendering time for different devices, and measure the size of messages transmited during distributed operation. Additionally, Section 3.1 described our user study which evaluated the split interface and altered prominence adaptation methods.

We demonstrate SUPPLE's versatility by exhibiting the wide range of different types of interfaces it has generated. Earlier in the paper, we presented a stereo controller (Figure 2), a fully functional email client (Figure 4), an interface to Amazon web services (Figure 5), a map-based interface (Figure 3), a controller for classroom equipement (Figure 6), and an adaptive print dialog box (Figure 9). The wide diversity of these applications demonstrates SUPPLE is capable of handling complex interfaces and rich data types (*i.e.*, the capability requirement).

[3] Both the factors comprising cost functions and the elements of the widget factory are described procedurally. As a result, appropriate class definitions may need to be submitted together with a rendering request.

	Print Dialog	**Map**	**Amazon**	**Email**	**Stereo**	**Classroom**
Lines of code	117	47	59	475	133	73

Table 2. Lines of code used to construct and manage the user interfaces for the applications presented throughout this paper and for the Classroom controller from [3].

	Print Dialog	**Map**	**Amazon**	**Email**	**Stereo**	**Classroom**
PC	0.31s	0.25s	1.2s	0.21s	1.5s	0.55s
PDA (local)	—	—	—	—	40s	29s
PDA (remote)	—	—	—	—	3.1s	2.5s

Table 3. Time required to render user interfaces on different platforms. The three conditions include interfaces being rendered locally on a desktop computer (PC) and interfaces rendered on a PDA either locally or on a remote solver server running on a desktop computer. In the last case the times include the communication overhead.

Comparisons of the code quantity or complexity among different approaches are often controversial. Yet, we feel it is useful to report on the amount of code[4] devoted to the description and management of the user interface for all the examples reported in this paper (*e.g.*, as a proxy for easy of use requirement). These numbers are reported in Table 2 and are for the programmatic (as opposed to the XML-based) encoding of the interfaces.

In service of the extensibility requirement, Table 3 reports the rendering times for different interfaces and different platforms. For interfaces running on a desktop computer, the generation times support fully interactive operation. PDA users can also experience fast interface generation times if they have network access to a remote solver. 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.

We have also measured the sizes of the messages that need to be exchanged in order to invoke a remote solver. 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 solver server an option, not only on WiFi networks, but also on slower Bluetooth or 3G cellphone connections.

7 Related Work

Researchers have investigated model-based user interface systems including automatic interface generation for many years, yielding impressive systems [15]. Unfortunately, none of the prior work meets our five requirements. Projects like the UIML [1], XIML [11] or the Ubiquitous Interactor [7] provide a device-independent way of representing various aspects of the user interface but ulti-

[4] Numbers were calculated using the Metrics plugin for Eclipse available at `metrics.sourceforge.net`.

mately all require that the application designer specify how abstract elements be mapped to concrete widgets on various target platforms.

The Personal Universal Controller (PUC) [5] comes closer to meeting our requirements as it provides a domain-independent language for abstractly describing user interfaces in terms of their functionality and it provides a set of rendering algorithms for a small number of different platforms. This system, however, is intended mainly for rendering interfaces for appliances as a replacement for traditional remote controls. Its rule-based rendering algorithms relies on specific domain knowledge and makes it inflexible even to the changes in the screen size of the device it runs on. XWeb [8], is even further limited by the fact that leaf widgets are pre-specified and only their layout is chosen dynamically.

8 Conclusions

This paper presents Supple — an automatic user interface generation toolkit that supports rapid prototyping and deployment of ubiquitous applications across different platforms. We make the following contributions that turned a research prototype into a practical and powerful tool:

– We provide a detailed description of the functional specification language for modeling user interfaces and we extend it with three new classes of features: explicit support for subtyping; new types for representing images, map locations and vectors; and the *alternatives* specification elements that give the designers greater control over different subsets of functionality to be presented on different classes of devices.
– We describe a new adaptation strategy, termed *split interfaces* that provides the benefits of allowing users fast access to frequently used functionality without needlessly disorienting them. This change required a fundamental extension to our rendering algorithm, enabling it to handle functional specifications which are directed acyclic graphs not trees.
– We support explicit customization actions, which allows both users and the designers to change the structure of the interface or to override Supple's automatic rendering choices.
– We use a distributed architecture, that enables user interfaces to be presented on remote device; it also allows impoverished devices with network connectivity to use remote interface solver servers for faster rendering of the interfaces. Our architecture also supports caching of the previously rendered interfaces for faster presentation and disconnected operation.
– Our evaluation demonstrates the feasibility of the practical deployment of Supple. Our user study (Section 3.1) supports the *split interfaces* as an effective and non-distracting method to adapt interfaces to user's tasks.

Supple satisfies the five desiderata listed in Section 1 and is a promising platform for ubiquitous interfaces. In order to fully evaluate Supple's impact, we are releasing it as an open source toolkit for public use.

In the future we plan to extend the specification language to allow drag and drop operations on platforms that provide that capability and to support more complex map-based interactions where additional interactive objects can be overlaid over the map. Finally, following [12], we plan to extend Supple to work with multiple modalities, including speech.

Acknowledgments We thank Don Patterson, Michael Cafarella and the anonymous reviewers for comments on the earlier versions of the manuscript. This research is supported by NSF grant IIS-0307906, ONR grant N00014-02-1-0932 and by DARPA project CALO through SRI grant number 03-000225.

References

1. M. Abrams, C. Phanouriou, A. L. Batongbacal, S. M. Williams, and J. E. Shuster. UIML: An appliance-independent xml user interface language. *WWW8 / Computer Networks*, 31(11-16):1695–1708, 1999.
2. L. Findlater and J. McGrenere. A comparison of static, adaptive, and adaptable menus. In *Proceedings of ACM CHI 2004*, pages 89–96, 2004.
3. K. Gajos and D. S. Weld. Supple: automatically generating user interfaces. In *IUI'04*, Funchal, Madeira, Portugal, 2004. ACM Press.
4. K. Gajos and D. S. Weld. Preference elicitation for interface optimization. In *Proceedings of UIST 2005*, Seattle, WA, USA, 2005.
5. J. Nichols, B. Myers, M. Higgins, J. Hughes, T. Harris, R. Rosenfeld, and M. Pignol. Generating remote control interfaces for complex appliances. In *Proceedings of UIST'02*, Paris, France, 2002.
6. 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 *UIST'02*, Paris, France, 2002.
7. S. Nylander, M. Bylund, and A. Waern. The ubiquitous interactor - device independent access to mobile services. In *CADUI'2004*, Funchal, Portugal, 2004.
8. D. R. Olsen, S. Jefferies, T. Nielsen, W. Moyes, and P. Fredrickson. Cross-modal interaction using XWeb. In *UIST'00*, pages 191–200, San Diego, California, United States, 2000. ACM Press.
9. M. Perkowitz and O. Etzioni. Towards adaptive web sites: Conceptual framework and case study. *Artificial Intelligence*, 118:245–276, 2000.
10. S. Ponnekanti, B. Lee, A. Fox, P. Hanrahan, and T. Winograd. ICrafter: A service framework for ubiquitous computing environments. In *Proceedings of Ubicomp 2001*, pages 56–75, 2001.
11. A. Puerta and J. Eisenstein. XIML: A universal language for user interfaces, 2002. unpublished paper available at http://www.ximl.org/.
12. D. Reitter, E. Panttaja, and F. Cummins. UI on the fly: Generating a multimodal user interface. In *HLT/NAACL-04*, 2004.
13. A. Sears and B. Shneiderman. Split menus: effectively using selection frequency to organize menus. *ACM Trans. Comput.-Hum. Interact.*, 1(1):27–51, 1994.
14. B. Smyth and P. Cotter. Personalized adaptive navigation for mobile portals. In *Proceedings of ECAI/PAIS'02*, Lyons, France, 2002.
15. P. Szekely. Retrospective and challenges for model-based interface development. In F. Bodart and J. Vanderdonckt, editors, *Design, Specification and Verification of Interactive Systems '96*, pages 1–27, Wien, 1996. Springer-Verlag.
16. D. S. Weld, C. Anderson, P. Domingos, O. Etzioni, K. Gajos, T. Lau, and S. Wolfman. Automatically personalizing user interfaces. In *IJCAI03*, Acapulco, Mexico, August 2003.