Automatic Generation of Device User-Interfaces?

Olufisayo Omojokun
Georgia Institute of Technology
omojokun@cc.gatech.edu

Prasun Dewan
University of North Carolina at Chapel Hill
dewan@cs.unc.edu

Abstract

One of the visions of pervasive computing is using mobile computers to interact with networked devices. A question raised by this vision is: Should the user-interfaces of these devices be handcrafted manually or generated automatically? Based on experience within the domain of desktop computing, the answer seems to be that automatic generation is not flexible enough to support a significant number of useful interfaces but requires substantially less coding effort for the interfaces it can create. We show that the answer is much more complicated when we consider networking of traditional appliances such as stereos and TVs. Using qualitative arguments and quantitative experimental data, we show that the manual vs. generated issue must be resolved based on: (a) not only user-interface programming and flexibility but also several other metrics such as space and time costs, binding time, and reliability (b) whether it is a graphical or speech based user-interface, (c) the size of the device user-interface, (d) whether the manually written user-interface code is available at the mobile computer or at a remote machine, and (e) the network bandwidth between the mobile computer and remote factory.

1. Introduction

One domain currently receiving a significant amount of research attention is networking arbitrary devices such as TVs, refrigerators, and sensors. It is attractive to create software user-interfaces on mobile computers to interact with such appliances, for several reasons:

Truly universal: Some traditional remote controls can interact with multiple devices such as TVs, VCRs, cable set-top boxes, and CD players. They are in fact called 'universal' remote controls, but have two important restrictions. First, a traditional universal control can interact with a fixed number of device instances. The amount of physical buttons and other controls on the remote determines this number. Mobile computers, on the other hand, can control arbitrary numbers of device instances. For example, mobile computers could allow security guards to control the lights in all current and future buildings in which they work. Second, a traditional universal control must provide buttons for the union of the operations among device types it can control, which can clutter it if the devices types share few operations. Therefore, universal controls typically support similar types of devices, that is, devices such as CD players, DVD players, and VCRs that share a large number of operations. Dissimilar devices such as fans and robotic vacuum cleaners require separate controls. A survey shows that 44% of households in USA have up to six remote controls [1]. A mobile computer can serve as a single control for arbitrarily different kinds of devices.

Automatic binding: Traditional universal remote controls require users to manually enter specific codes for the device instances they wish to use. For instance, universal remotes for controlling home entertainment devices require users to look up the manufacturer codes of their devices (TVs, VCRs, etc) and enter these codes on the remote. This design does not create a serious problem when the number of devices is small, but would have a significant drawback in a world with ubiquitous computing. Since mobile computers are intelligent, they can automatically bind themselves to arbitrary device instances through a discovery process [2-7].

Truly remote: Since IR signals cannot pass through walls, some traditional "remote" controls only allow users to control devices in the vicinity of a user. X10 remote controls are based on radio signals, so they are not limited by walls. However, these signals can only travel a few feet. A mobile computer can interact with a networked device over the Internet. Thus, it can be used to control a device from an arbitrary location. For example, a mobile computer can allow a person on vacation to deactivate a security system at home so that a neighbor can freely enter the house to feed fish in an aquarium. If the security system ever needs troubleshooting, a technician at the manufacturer's site could use a mobile computer to possibly fix the device without having to visit the owner's home.

Beyond physical user-interfaces: Perhaps a more fascinating reason for using mobile computers to interact with networked devices is that it is possible to create software user-interfaces for them that are more sophisticated than the conventional physical user-interfaces offered by traditional appliances and their controls [8]. Unlike conventional user-interfaces, they can group related controls into overlapping tabs [8, 9], display state [8, 9], allow offline editing, provide consistency [10], and be customized to the habits of their users [9].

Not surprising then, several systems today offer software-based user-interfaces, which include: Palm/Pocket-PC IR programs [1, 11], HP's Cooltown [12], IBM's Moca[2], and Websplitter[13], Microsoft's UPnP [5, 14], Sun's Jini[6, 15], CMU's PUC [8] and UNIFORM [10], Cornell's Cougar [16], Swedish Institute's Universal Interactor [17], Media Lab's UI on the Fly [18], Berkeley's TinyDB [19], [20], and DAMASK [21], Stanford's ICrafter [22], U. of Washington's SUPPLE [9], and PARC/Georgia Tech's Speakeasy/Obje [23]. In most of these systems, the user-interfaces are manually implemented. A few systems [8, 9, 17, 18, 20, 22], on the other hand, explore the intriguing idea of automatically generating these user-interfaces. While this idea is new in the domain of mobile/device computing, it has been explored for over three decades in the realm of desktop computing [24-27]. The lessons from desktop computing tell us that automatic generation is not flexible enough to support a significant number of useful interfaces but requires substantially less coding effort for the interfaces it can create. The question we try to answer here is: do these lessons also apply to the area of mobile/device computing? Naturally, the answer depends on the kind of device user-interfaces and generation algorithm we address. We consider software user-interfaces of individual devices and do not address interfaces for dynamic compositions of devices [9, 22, 28, 29]. Generation algorithms can be classified into those using heuristics to meet high-level goals such as uniformity [10] and low usage of screen real estate [9], and those that are based on user-provided specifications. We address specification-based generation. Thus, our initial answer to the question above ignores heuristics-based automatic generation and dynamic-device composition.

Intuitively, there are several reasons why the manual/automatic question must be re-examined in the context of device user-interfaces. Given a networked device, its user-interface must be implemented on each of the large number of the continuously evolving mobile computers that may be used to interact with it. Previous papers have hypothesized that this effort would be high, and have used this hypothesis as the motivation for (specification-based) generation in this domain. In addition, these interfaces seem to be simpler and have less variety than desktop user-interfaces, consisting mainly of rectangular arrangements of buttons and simple widgets. Thus the inherent lack of flexibility of generation could become less of an issue. On the other hand, because they are simpler, the automation provided by generation may become less of an advantage. Devices and mobile computers are much less powerful than desktop PCs – thus we must consider space and time costs. Some of these metrics should depend on whether the interfaces consist only of commands or includes both commands and state, as the latter are more complex. Complicating the issue is the fact that mobile devices have limited screen space and may be used by users whose hands are occupied on some other task, making speech user-interfaces (SUIs) a practical alternative to GUIs [21-23, 30-32]. Thus the manual/automatic question must be answered also for SUIs. As the devices are networked, the impact of communication delays on the choice of the approach needs to be explored.

At first glance, it seems we must also consider usability. However, this is a quality of individual user-interfaces and not the framework used to implement them – both the manual and generation approaches may be used to create interfaces that are both easy and hard to use. We consider system rather than usability issues as we are addressing frameworks rather than individual applications. In our experiments, we have included the only two device interfaces known to us that have been shown (by others) to be easy to use [9, 30]. The other interfaces we consider are based on a systematic way of converting commercial physical interfaces to software user-interfaces. Another important metric we ignore is the interoperability effort required to port a user-interface for one mobile computer to another kind of mobile computer[21]. The impact of the approach on energy costs is also left as future work.

The paper consists of both qualitative arguments and quantitative experimental data. It is a substantial update and extension of our previous workshop paper [33]. More details on this issue can be found in the first author's thesis [34]. The rest of the paper is organized as follows. We first present our models of the generation and manual frameworks. We then outline the metrics we used in our analysis and describe our experimental setup and benchmarks. The bulk of the paper compares the two approaches based on these metrics. Finally, we present conclusions and directions for future work.

2. Model of Manual/Automatic Generation

Figure 1 shows a general architecture that abstracts existing systems for deploying device user-interfaces on mobile computers that applies to both the automatic and manual approaches. Device objects encapsulate the functionality of actual physical devices. They contain methods for invoking commands on devices and viewing device state. Device advertisers publish information about devices and references to them within a given network or physical space. They are accessed by device discoverers on mobile computers. Device advertisers may run on the same host as that of the device objects or on a separate machine. User-interface deployers on mobile computers, using device references, deploy the actual user-interfaces for interacting with device objects. Here we consider only the issue of whether the deployed user-interface is created manually or automatically. Thus, we do not consider other problems such as secure device discovery and assume state-of-the-art solutions to them.

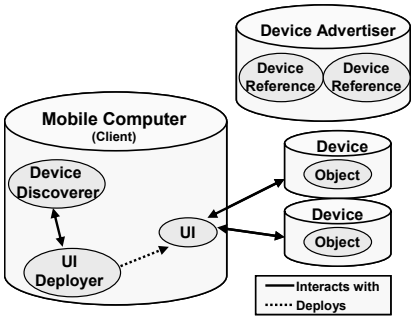

Figure 1. A general architecture for deploying device user-interfaces.

In many manual approaches such as Nevo [35] and OmniRemote [11], a predefined implementation of the user-interface resides on the mobile computer. Others such as [2, 5, 12, 15], offer an alternative approach in which the code resides on a remote computer. Thus, the manual approach has two variations, *local* and *remote*, distinguished by where the predefined user-interface is located. We consider both variations. Both manual implementations and generator specifications can be created interactively or programmatically. We assume both are created programmatically.

(Specification-based) generators create default user-interfaces for devices, which can be overridden by high-level specifications. There is no well-defined model of such generators, in the desktop or device-mobile computing domain. Therefore, here, we identify such a model for the device-mobile computing domain. It includes: (a) features of generators in this domain published in previous papers, and (b) features identified by us to completely generate published interfaces produced by these generators. We give enough details so that the readers can understand the programming-cost numbers and specify the user-interfaces shown here.

Our generator is given a description of the state properties and operations of a device. This description may be automatically derived from the object coding the device, or it may be created manually using an external, language-independent, description – our analysis does not distinguish between these two approaches as it does not measure the cost of creating an external description. We restrict operation parameters to simple types, as complex types do not appear in device user-interfaces, but allow properties to be of predefined and user-defined types as device objects can be hierarchical. Moreover, properties and operations can be collected into hierarchical user-defined view groups [30] to allow the view and dialogue structure to be independent of the device structure. We allow view groups to overlap, thereby allowing the use of alternative GUI-views and SUI-dialogues to invoke an operation or modify a property. Let us first consider GUI generation.

An operation is mapped to a button. Any parameters of the operation are collected using a dialogue box. A property of a predefined type is mapped to one of a set of widgets associated with the type. Each predefined value can be mapped to a text box displaying the textual representation of the value. In addition, a Boolean value can be mapped to a checkbox, an integer value to a slider, and an enumeration to a combo-box (Figure 2, Band property) or radio button (Figure 3, Mode property). Finally a property associated with increment and decrement operations can be mapped to an *IncDec* widget that consists of a text-widget to display the property value and buttons to invoke the two operations (Figure 2, widget displaying current station). A user-defined type or view group is mapped to a panel containing the display of its properties and operations. These panels can be arranged in tabs (Figure 2 and 3). A user-specified attribute of the property/view group selects from the alternate widgets and containers defined for it. In addition, we assume attributes that determine the label and position of a property and operation, and whether a label is displayed. The positions of properties and operations are used when they are linearly arranged, as in different tabs.

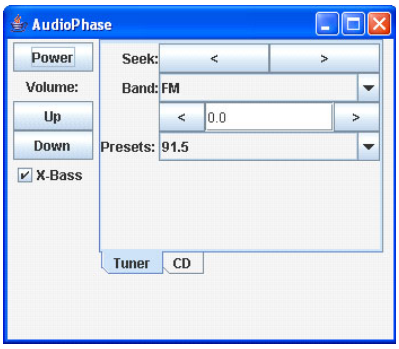

Figure 2. PUC Stereo UI

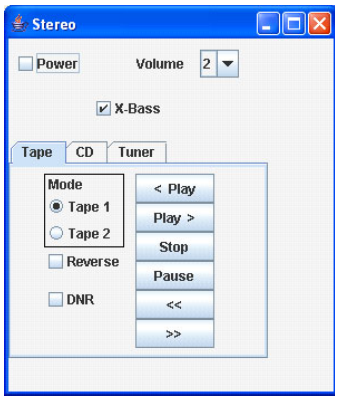

Figure 3. Supple Stereo UI

To accommodate the two-dimensional displays of devices, we define a special layout that extends the grid layouts of toolkits in the following ways: (a) the placement of components (operations/properties) is based on user-specified row and column attributes rather than the sequence in which they are added to the parent container; (b) a component may be associated with (text or iconic) labels displayed above, below, left and right of it, (c) for positions in the grid not associated with a component or label, a filler label is displayed whose content (text/icon) is specified by an attribute, (e) and, depending on another attribute, the rows in a grid may have their preferred sizes or be made equal size by stretching rows smaller than the largest row.

By default, we assume that the generator: (a) creates checkboxes and combo-boxes for Boolean and enumeration values and textboxes for other primitive values, (b) creates non-tabbed sub-panels for composite properties and view groups, (c) arranges operations and properties in separate grids ordered by their names, (d) uses the name of a component in the programming interface as a label, and (e) uses a single space character for the filler label. Figure 4 shows a default display created when no attributes are specified. These defaults are necessary to interpret the numbers we give for the lines of code required to specify various user-interfaces.

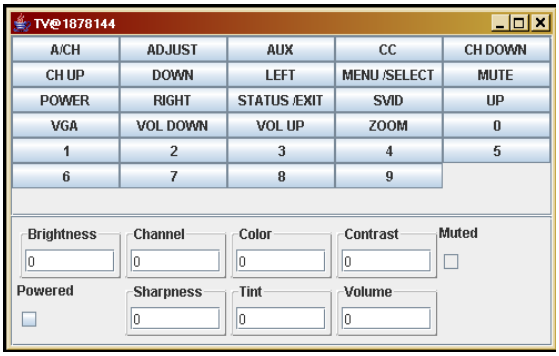

Figure 4. A TV GUI using all defaults.

It is possible to override defaults globally or for an individual property and operation. For example, in Figure 3, a global specification says that rows are not of equal size. As a result, operations, properties and view groups such as “Power”, “Volume” “X-Bass”, and “Tuner” that do not override this default have their preferred sizes. On the other hand, the column containing “< Play,” “Play >” and other operations has rows of equal sizes. This is because the view group containing them overrides the value of this attribute.

Our model of a speech generator also combines and extends existing techniques for mapping objects to dialogues. Each (parameterized) operation is associated with a (parameterized) command and each composite property/view group with a sub-dialogue. Commands are also provided to set and get simple properties. An attribute determines the spoken name of the operation/property, which by default is the same as the identifier used to name the component in the programming interface. In the system-initiative mode, the system prompts users for the alternative choices, in order of the values of their position attributes. In the user-initiative mode, the user simply gives the complete command. As view groups can overlap, all operations and properties with distinct names are put in a top-level view group so experienced users do not have to navigate through submenus in the user-initiative mode. (As mentioned in [30], in this domain, duplication of names does not occur.) Elements of the top view group that are also in some other child group are not presented as options in the system-initiative mode to keep menus short.

We assume that the GUI and SUI generator models are built on top of GUI and SUI toolkits, which

themselves provide customization features such as determining the alignment of labels and the width of text- and combo- boxes. We assume these generators allow toolkit customization features to be used by the developers. Exercising them has the same cost in both the manual and generation approaches, so we ignore them in our discussion and implementations.

3. Evaluation

The evaluation metrics we used extend the ones used for desktop computing and take into account the special features of this domain:

1) *User-Interface Flexibility* – range of user-interfaces that an approach can support.
2) *Programming Costs* – amount of code required to define (implement/specify) a user-interface.
3) *Efficiency* – time and storage space costs of an approach.
4) *Device Binding Time* – time a client must learn about (or bind to) a device in order to deploy a user-interface for it.
5) *Deployment Reliability* – the level of guarantee an approach offers in deploying a user-interface.

The first three metrics are applicable to both desktop and device user-interfaces. Efficiency takes into account the fact that devices and mobile computers have restricted computing power and devices have restricted memory. Device binding time is important in our domain because the functionality and user-interface are distributed on two separate machines, which must be bound to each other. Deployment reliability determines the likelihood an approach provides a user-interface for a network device. These metrics allow us to make a mix of qualitative and quantitative comparisons of the various approaches. As there is no previous work on quantitative comparative evaluation based on these metrics, we discuss below in some depth our choice of benchmarks used for such comparisons.

Some of the quantitative comparisons require performing experiments with real networked devices. Thus, we networked six actual devices of different types: a Phillips TV, JVC VCR, Sony A/V Receiver, Panasonic DVD Player, Hitachi Projector, and lamp. The first author owns all of the devices except the projector, which is found in a conference room inside our department's building.

For each device, we created a Java RMI (proxy) object representing its functionality on a desktop PC. Each object has a programming interface that consists of a method for each command (or button) found on its associated device's traditional remote control and Java Bean getter and setter methods for each unit of state. When invoked, a method executes code that sends a signal to its corresponding (actual) device to perform the associated command. The desktop PC has an IR and X10 radio module connected to its serial ports for sending these signals. The IR module has a record/playback facility, which we used to store the signals of all of the commands found on the TV, VCR, receiver, DVD player, and projector remote controls. Each method invocation simply replays the recorded signal of its associated command—it does not return any value or require any parameters. The lamp is the only device that we networked using the X10 protocol.

Our next step was to identify a set of diverse but "realistic" benchmark user-interfaces for our experiments. We created four kinds of device user-interfaces:

Direct Mimicking: These user-interfaces directly mimic traditional remote controls by mapping each physical button to a soft button, ensuring that groupings of the buttons and their labels are maintained. The top panel of Figure 5 is an example of such an interface.

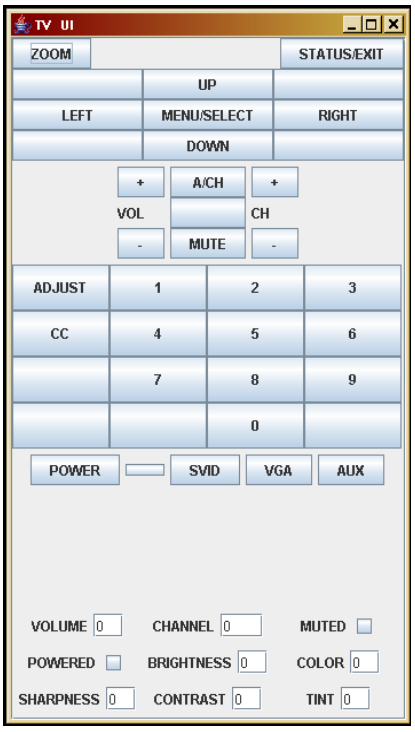

Figure 5. A command and state TV GUI

State-containing User-Interfaces: To expand the applicability of our results, we wished to also investigate user-interfaces that incorporate state. Since traditional remote controls do not display state, there

are no clear-cut examples to directly mimic. However, we can make a logical derivation of such GUIs by searching the on-board panels and remote controls of each device for status information and extend the standard grid-based layout of buttons to display state. The bottom panel of Figure 5 illustrates this derivation by displaying the TV's handcrafted GUI with state. This GUI displays the device's properties using widgets that can appropriately display their values. The values of Boolean properties are displayed using checkboxes while the values of strings and numeric types are displayed using textboxes. To illustrate, the TV's 'powered' property value is displayed by a checkbox. An unchecked box means that the TV is off, otherwise, it is on.

Beyond Physical User Interfaces: Even though traditional appliance panels and their remote controls have been a principal means for interacting with devices for many years and their interfaces have not changed much over these years, as mentioned in the introduction, it is intriguing to explore more sophisticated software device user-interfaces. We used the only two examples of such interfaces available to us, both of which have been evaluated for usability in the lab – the PUC AudioPhase user interface (Figure 2) and the Supple Stereo user-interfaces (Figure 3). These were good candidates as they provide very different user-interfaces for essentially the same set of operations and properties and make extensive use of view groups. In the PUC case, we could not find the tape controls in any published work, so we included only the tuner and CD controls in the user interface. As we did not have the associated physical devices, the proxy objects we created for them did not, in fact, control real devices.

Speech User-Interfaces: We also derived the speech user-interfaces of the six real (two simulated devices) from their corresponding state-containing (beyond-physical) graphical user interfaces by creating a sub-dialogue for each view group, allowing users to speak out the text of the widget used for the operation or property to invoke/query it, supporting both user and system initiative, and in the system initiative mode, ordering the choices by the associated graphical position attributes.

The nature of the user-interface affects only the results of the flexibility and programming costs metrics –as other metrics are either evaluated qualitatively or, in the case of deployment time, is a function primarily of the number of operations and properties and the mode of user-interface (SUI/GUI). Even the flexibility and programming-cost metrics are somewhat independent of the exact user-interface. In particular, the exact position or name of a speech prompt or a user-interface widget is not important – what matters is that it is explicitly specified and can be supported by the generator model. While the picky reader may quibble with the exact set of user-interfaces we have chosen, it is important to keep in mind that we have tried to do better than the well-accepted approach in other domains of picking random samples when established benchmarks do not exist.

Our manual implementations of the user-interfaces were done using Swing. For an implementation of the generator model described previously, we extended an existing Java-based generator, called ObjectEditor, which had previously been used in teaching and research. As it is a general-purpose generator, it includes several features not in our model generator, such as programmer-defined parameters. For an implementation of our SUI-generator model, we created a new tool. Neither tool requires an external description of the appliance object, they create the description automatically using runtime analysis or reflection of object signatures. They identify state properties described by signatures adhering to the Java Bean conventions. Signatures that are not used to export state properties describe operations. As a result, the performance numbers we give include the reflection cost.

We are finally ready to start evaluating the approaches based on the dimensions given above.

User-Interface Flexibility*:* The manual approach, can, of course, support arbitrary user-interfaces, so the flexibility comparison requires the degree to which the user-interfaces in this domain can be supported by the generation approach. This question has been partly addressed in papers describing some existing generators by showing that the generators can create an interesting set of usable device user-interfaces. However, these papers do not address the question of whether the generators can create (device) user-interfaces not designed by their authors – for example, user-interfaces created by other generators. We contribute to this issue by determining what fraction of the diverse benchmark user-interfaces can be supported by the generator model described above.

We found that the generator model supports all of our target speech and graphical user interfaces. The fact that it supports all of the speech user-interfaces is not that surprising. We did not have existing candidates for them, so, as mentioned above, we created them based on the corresponding graphical user-interfaces and general principles about their design [30], which are embodied in the generator. Moreover, a cursory look at speech user-interfaces of web services such as airline systems shows that they follow these principles. GUIs show much more variety, so it is more significant that the model supports all of the target interfaces. Our model, in turn,

was designed for these interfaces, so another way to state this observation is to say that we were able to create a model to support all of these interfaces. When designing the model, it was necessary to look only at two user-interfaces, one command-based remote control interface (Figure 5, top panel) and one beyond-physical user interface (Figure 3). Once these were supported, the other user-interfaces required no additional features in the model. This implies that the model has applicability far beyond the benchmark user-interfaces. Of course, in the future, there could be more sophisticated device user-interfaces that are not supported by (minor tweaks to) our model. Such user-interfaces would probably be "beyond physical user-interfaces" that are very different from the ones invented so far. It is in this space where the tradeoff between the automation and user-interface flexibility of generation probably exists in the device/mobile computing domain. We expand below on this tradeoff by evaluating the programming costs involved in the different approaches.

Programming Cost*:* Previous research, in the desktop and device domains, has measured the cost of manually implementing desktop GUIs [36, 37] and the cost of specifying desktop GUIs [25] and device GUIs [38]. Our contribution here is a head-to-head comparison of the programming costs of the two approaches using the target set of GUIs and SUIs. In the absence of a better measure, like the previous works referenced above, we use number of lines of code/specification needed to implement/specify the user-interfaces as a measure of the programming cost. We omit comments in our count.

Table 1 shows the results. As can be expected, the amount of code is directly proportional to the size of the device in terms of the number of commands and properties. Interestingly, the speech interfaces needed less code/specification than the corresponding GUIs. The specification code was not negligible because all operations/properties needed positions and many of them needed labels different from their names. Even then, the generation approach offers 3-15 times fewer lines of code/specification. The benefit is greater for devices such as a lamp with a small number of operations/commands because in the manual approach there is a fixed cost of interfacing with an underlying toolkit and implementing base application-independent functionality such as transitioning between user-initiative and system-initiative dialogues. These tasks are automated by the generator. Assuming that, on average, a line of specification does not require more effort than a line of code, it seems generation is a big winner. However, the absolute values of the numbers for the manual approach are small given that many commercial programs are thousands, if not millions of lines of code. Still, the manual approach can be considered to impose a significant overhead given that literature reports programmer productivity to be 250-550 lines of code per month [39]. Another factor that muddles the issue is the cost of learning new software. Graphics toolkits are standard and popular, while GUI generators are not. Therefore, it may not be worthwhile to learn a user-interface generator. On the other hand, most are probably not familiar with speech APIs, which tend to be non-standard. Learning to use a generator will probably take much less time. Moreover, the speech and graphical generators in our model provide an integrated specification mechanism, which can be learned once to support both modalities. Finally, given a device, the productivity gain of the automation approach applies to each kind of mobile computer on which a user-interface of the device is created, assuming no code sharing occurs (in both the manual and automatic approaches) among the different kinds of mobile computers. (Automatically or semi-automatically porting a user-interface implementation to multiple devices is still an area of research [21]. In summary, as the productivity gain occurs without loss of flexibility, the generation approach is a clear but not big winner if we do not consider other metrics.

Table 1. Number of lines of UI code used for each device

Device	# of cmds	# of props	# of lines of UI code					
			GUIs				SUIs	
			Command-only		Command and State		Semi-automatic generation	Manual
			Semi-automatic generation	Manual	Semi-automatic generation	Manual		
Lamp	4	2	4	94	8	110	6	102
Projector	20	4	23	254	31	283	24	138
TV	29	9	25	321	36	385	33	142
VCR	46	5	40	247	47	288	46	172
DVD Player	38	14	38	316	54	414	52	168
Receiver	42	22	42	287	66	428	64	176
PUC Stereo					115	367	61	
Sup. Stereo					137	546	95	

Deployment Reliability:

The metric that makes the most compelling argument for generation is deployment reliability. So far, we have assumed that a user-interface specification or implementation of each device has been created for each mobile computer. In practice, this may not be the case. Our generation model, on the other hand, always guarantees a user-interface, albeit, in the case of GUIs, a less than ideal one. Assuming a generated user-interface such as the one shown in Figure 4 (which, recall, has all the buttons and widgets of the target user-interface) is better than no interface, generation is a more reliable approach. We assume here that a generator for a mobile computer is shipped with the computer.

Binding Time: When a manual user-interface has been implemented for a mobile computer, we must address binding time. There are two times a mobile client can learn about (or bind to) a device so that it can display the user-interface for it. In early binding, users must manually install the user-interface code for devices they expect to use in the future on their clients. Consequently, they will not be able to interact with a device if its user-interface code is not already stored on their clients. In late binding, no pre-installation is necessary. Instead, the user-interface for a device is automatically deployed at interaction time and thus requires no user anticipation. Therefore, users can interact with arbitrary devices. The local-manual approach inherently supports early binding, and the remote-manual and generation approaches support late binding. Thus, based on this factor, the comparison between the automatic and manual approaches depends on the variation of the manual approach, with the remote approach performing better.

Space Cost: On the other hand, a disadvantage of the remote approach is that some networked devices have limited storage, which in the remote approach, must be used for storing the user-interfaces of all possible mobile devices that may interact with them. A document on UPnP estimating the power of networked devices states that: "typically, they are based on a low-cost micro controller, ASICs and some 200-1000 k bytes of RAM and Flash memory [14]." To understand the space requirements of the target user-interfaces, we measured the footprint (in bytes) of the user-interface code we wrote for each of the target devices. Table 2 shows that the remote approach is practical only if networked devices have substantially more memory than the UPnP estimate. It also motivates a variation of the remote approach in which the user-interface code is downloaded from a special user-interface server. We assume a mobile computer has enough space to store manually implemented user-interface code for a plethora of devices or the generator.

Table 2. Amount of space consumed by each device's handcrafted UI code.

Device	Space Consumed (bytes)		
	GUIs		SUIs (manual)
	Command-only	Command + State	
Receiver	9,728	11,737	5945
DVD Player	9,216	11,086	5723
Lamp	2970	3,516	3827
Projector	6753	6,958	5375
TV	8,704	9,377	5442
VCR	9028	10,064	5845

Interaction Time: So far, the generation approach has fared better or at least as well as the best manual approach. However, by definition, it should have larger time costs than custom code – so the question here is how much larger are these costs. One important component of these costs is interaction time – the time it takes to invoke an operation on the remote device. Somewhat surprisingly, our experiments showed a negligible difference between the manual and automatic approaches for both SUIs and GUIs, despite the fact that our generator uses reflection and other forms of indirection to invoke operations. We consider a time difference significant if it is more than 50ms, which is the delay noticeable by users [40].

Deployment Time Another important component of time costs is the time it takes to deploy the user-interface. Previous work has shown that on an iPAQ this time can be as large as 40 seconds for the user-interface of Figure 3 when the generator is both specification- and heuristics-based [9]. However, it does not compare this cost with the two variations of the manual approach. Qualitatively, generating a user-interface can be expected to take longer than starting a local copy of manually written user-interface code. Similarly, deployment time can be expected to be more in the remote-manual rather than local-manual approach. However, it is not possible to make qualitative arguments: (a) comparing the generation and remote-manual approach or (b) about the extent by which the local-manual approach wins over the other two approaches. Therefore, we performed experiments comparing the three approaches with respect to the GUI and SUI interfaces of the six devices. We used two different kinds of mobile computers: (1) an old laptop {Windows 2000 OS, 733 MHz Pentium, 128MB} and (2) an iPAQ Pocket PC {SavaJe Java-based operating system, 206MHz StrongArm, 32MB}. The old laptop was meant to represent future palmtop computers. We considered three kinds of network

connections: dialup (50Kbps), wireless (1Mbps), and wired (11Mbps) LAN. The dialup case was meant to represent cell-phone connections, some of which have close to dialup speeds.

Because of space limitations, we discuss the results of only a few of these experiments (Figure 6) – more details are given in [34]. All the generation times assume that the generator is preloaded since it is a general purpose application. A mobile computer will execute the same generation algorithm to create different device user-interfaces throughout a user's course of interactions. It does not make sense to consider the case of always keeping manually written user-interfaces in memory since functionality (or code) from one user-interface implementation cannot be used to help deploy another. Somewhat surprisingly, we found that preloading the generator can reduce the deployment time by an order of magnitude. In our experiments with the receiver, Figure 6a shows that on the iPAQ, the remote-manual approach is 1.57 times slower than the local-manual approach, while generation is 3.0 and 1.90 times slower than the local-manual and remote-manual approaches respectively. The laptop results follow this trend, but have significantly lower deployment times, thus showing that processing power is an issue for both the manual and generation approaches. The generation, remote-manual, and local-manual approaches respectively took 3.78, 4.4, and 2.8 times longer on the iPAQ than on the laptop. It even took less time to generate the GUI on the laptop than to deploy one on iPAQ using the local-manual approach (Figure 6a). Thus, the deployment times were significantly larger in generation than in other two approaches, but were much smaller than the ones reported in [9] even though our implementation makes extensive use of layering and dynamic reflection. The better times can be attributed to the fact that unlike [9], our implementation does not use heuristics. Moreover, in the experiment reported in [9], the generator may not have been preloaded.

***UI Size*:** We found that these times are a function only of the size of the interface and not its layout. For example, on the iPAQ, in the generation approach, the receiver's command- and state- based GUI takes five times longer than that of the lamp.

Network Speed The above results assume a wired LAN connection. We also measured the effect of network speed on the relative deployment times (Figure 6b). As can be expected, the remote-manual approach is sensitive to the network speed. Perhaps less intuitive, so are the local-manual and generation approaches. The local-manual approach must remotely download device state before the user-interface is ready, which causes, in this example, the deployment in the wireless and dialup connections to respectively take 1.21 and 16.19 times longer than the wired LAN connection. The

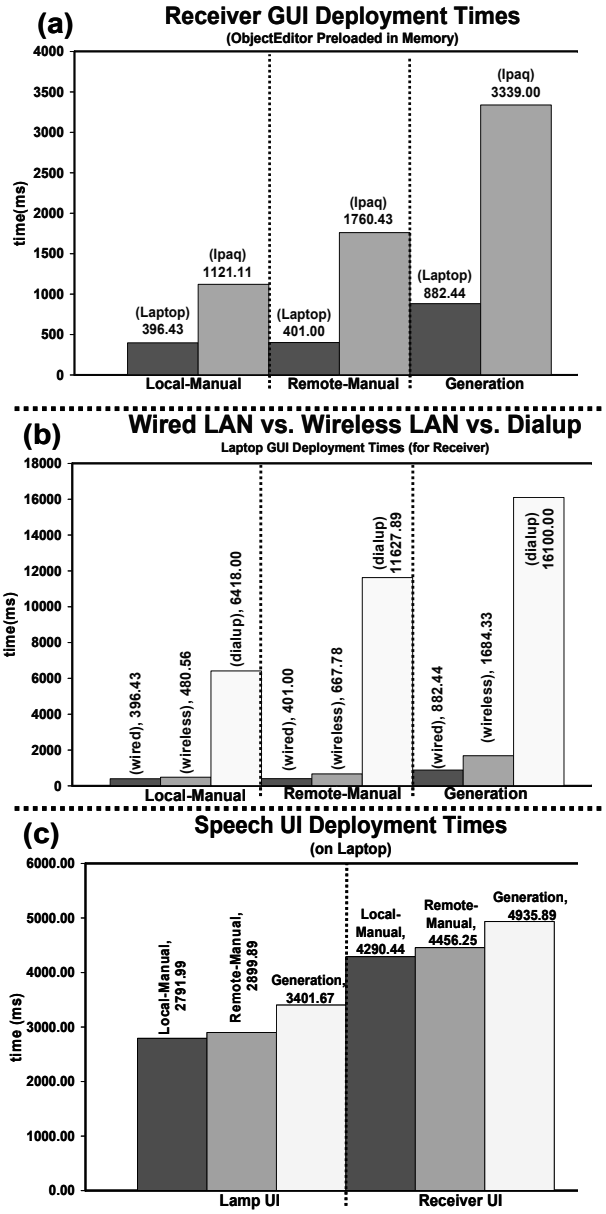

Figure 6. (a) Receiver command and state GUI based times (laptop and iPAQ with a wired LAN connection); (b) Receiver command and state GUI based times (laptop with different LAN connections); (c) Receiver and lamp command-only SUI based times (laptop with a wired LAN connection).

proportional differences are greater under the remote-manual approach because it must additionally download code for an entire user-interface—not just property values. Specifically, the wireless and dialup

connections respectively take 1.67 and 30 times longer than the wired LAN connection. The generation approach is also more network-dependent than the local-manual approach. Before downloading state, a generator must first download device descriptions from which it creates user-interfaces. Thus, just going from a wired to wireless connection can double generation time. Finally, generating with the dialup connection takes 18.24 times longer than with the wired LAN connection.

SUIs: SUI deployment follows the same trend as GUI deployment (Figure 6c). In the SUI case, however, the differences between the three approaches were not as significant. Our results also show that speech user-interface deployment takes much longer than GUI deployment no matter the approach—implying that SUI deployment is a more resource intensive process than GUI deployment. Deploying a local-manual lamp (command-only) SUI (2791.99 ms) takes over twice as long as generating a receiver (command-only) GUI (505.50 ms) when using the wired LAN connection. Further, generating the receiver SUI using the same network connection actually takes over four seconds. This explains why remote downloading and generation does not contribute as much to the deployment time in the SUI case.

4. Conclusions and Future Work

The novel aspects of our work can be described at various levels of detail. The most abstract message is that the question of whether user-interfaces should be automatically generated must be revisited in the context of device/mobile-computer interaction. The next-level message consists of the set of important metrics we have found to matter in making this decision: user-interface flexibility, programming overhead, space costs, deployment time, device-binding time, and deployment reliability. Only the first two matter in desktop computing. A related contribution is the set of factors we found that make a significant difference to these metrics, which include: the network speed, device complexity, processing power of mobile computer, the type of user-interface (command-only GUI, command- and state- based GUI, and SUI), and whether the generator is preloaded and is domain-specific. Another important contribution is the detailed model we have provided for specification-based SUI and GUI generation.

A more detailed contribution is the comparison of (semi-automatic and automatic) generation and (local and remote) manual approaches along each of the evaluation dimensions. These observations imply that each of the three approaches, local-manual, remote-manual, and automatic generation, has a unique benefit. The local-manual approach offers the lowest deployment time when early binding is acceptable. The remote-manual approach offers the best deployment time when late binding is required and device space cost is not an issue. Automatic generation simultaneously offers the maximum user-interface flexibility, lowest programming overhead, maximum deployment reliability, and late binding. A single computer could use multiple approaches and the choose the appropriate one based on the scenario – for example, fully automatic for new devices and local-manual for frequently used old devices.

The question posed by this paper is one that is often debated informally, and ours is simply an initial-cut at a systematic analysis of this issue. It would be useful to extend the: (a) set of target devices in the domain examined by including, for example, thermostats, sensors, and digital music streamers, (b) target domain by including, for instance, device user-interface personalized to a user, (c) the mobile computers used in the experiments by including, for instance, cell phones, (d) the set of metrics by including, for instance, maintenance costs, and (e) the set of approaches by including a generation approach that retargets existing user-interfaces [10, 20] rather than generating them from scratch, composes devices [22, 28, 29], and supports heuristics and remote generation [9].

5. Acknowledgements

This research was funded in part by IBM, *Microsoft* and NSF grants ANI 0229998, EIA 03-03590, and IIS 0312328.

6. References

[1] *Remote Possibilities*, in *USA Today*. 2000.

[2.] Beck, J., Geffault, A., and Islam, N. *MOCA: A Service Framework for Mobile Computing Devices*. in *International Workshop on Data Engineering for Wireless and Mobile Access*.

[3] Czerwinski, S., et al. *An Architecture for a Secure Service Discovery Service*. in *ACM MobiCom 1999*.

[4] Guttman, E. *Service Location Protocol: Automatic Discovery of IP Network Services*. in *IEEE Internet Computing*.

[5] Larsson, B.C.a.O., *Universal Plug and Play Connects Smart Devices.* WinHec 99 White Paper (http://www.axis.com/products/documentation/UPnP.doc).

[6] Sun Microsystems, I., *Jini technology architectural overview*: from http://www.jini.org, and Jini network technology http://www.sun.com/jini/).

[7] J. Beck, A.G., & N. Islam, *MOCA: A Service Framework for Mobile Computing Devices.* Proceedings of the International Workshop on Data Engineering for Wireless and Mobile Access.
[8] Nichols, J., et al. *Generating Remote Control Interfaces for Complex Appliances*. in *ACM Symposium on User Interface Software and Technology*. 02. Paris.
[9] Gajos, K. and D.S. Weld. *SUPPLE: Automatically Generating User Interfaces*. in *IUI*. 04.
[10] Nichols, J., B.A. Myers, and B. Rothrock. *UNIFORM: Automatically Generating Consistent Remote Control User Interfaces*. in *Proceedings of CHI'2006*. 2006.
[11] *OmniRemote*.
[12] Hewlett-Packard-Corporation, *Cooltown*.
[13] Han, R., V. Perret, and M. Naghshineh. *WebSplitter: A Unified XML Framework For Multi-Device Collaborative Web Browsing*. in *Proceedings of ACM Computer Supported Cooperative Work*. 2000.
[14] Schlimmer, J., *ChangeDisc:1 Sample Service Template For Universal Plug and Play Version 1.0*.
[15] Community, J. *The ServiceUI Project: UI Factories*. in *http://www.artima.com/jini/serviceui/UIFactories.html*.
[16] Bonnet, P., Gehrke, J., Seshadri, P. *Querying the Physical World*. in *IEEE Personal Communications*. 2000.
[17] Stina Nylander and M. Bylund. *The Ubiquitous Interactor: Universal Access to Mobile Services.* in *HCII*. 2003.
[18] Reitter, D., E. Panttaja, and F. Cummins. *UI on the fly: Generating a multimodal user interface.* in *HLT/NAACL*. 04.
[19] Madden, S.e.a. *The Design of an Acquisitional Query Processor for Sensor Networks*. in *SIGMOD*. 2003.
[20] Hodes, T. and R. Katz, *Composable Ad Hoc Location-Based Services For Heterogeneous Mobile Clients.* Wireless Networks, 1999. **5**: p. 411-427.
[21] Lin, J. and J.A. Landay. *Damask: A Tool for Early-Stage Design and Prototyping of Multi-Device User Interfaces*. 2002,.
[22] Ponnekanti, S.R., et al. *ICrafter: A Service Framework for Ubiquitous Computing Environments*. in *Ubicomp 2001*. 2001. Atlanta.
[23] Edwards, W., et al. *Recombinant Computing and the Speakeasy Approach*. in *Mobicom 2002*. 2002.
[24] Olsen, D.R. and E.P. Dempsey, *SYNGRAPH: A Graphical User Interface Generator.* Computer Graphics, July 1983. **17**(3): p. 43-50.
[25] Dewan, P. and M. Solomon, *An Approach to Support Automatic Generation of User Interfaces.* ACM Transactions on Programming Languages and Systems, October 1990. 12(4): p. 566-609.
[26] Sukaviriya, P., J. Foley, and T. Griffith. *A Second Generation User Interface Design Environment: The Model and Runtime Architecture*. in *Proceedings of Human Factor in Computing Systems: INTERCHI'93*. April 1993.
[27] Szekely, P. *Retrospective and Challenges for Model-Based Interface Development*. in *in 2nd International Workshop on Computer-Aided Design of User Interfaces*. 1996.
[28] Omojokun, O. and P. Dewan. *A High-level and Flexible Framework for Dynamically Composing Networked Devices*. in *Proceeding of 5th IEEE Workshop on Mobile Computing Systems and Applications*. 2003.
[29] Nichols, J., et al. *Huddle: Automatically Generating Interfaces for Systems of Multiple Connected Appliances*. in *Proc. UIST '06*. 2006.
[30] Jeffrey Nichols, B.M., Thomas K. Harris, and S.S. Roni Rosenfeld, Michael Higgins, Joseph Hughes. *Requirements for Automatically Generating Multi-Modal Interfaces for Complex Appliances*. in *Proc. ICMI*. 02.
[31] Lemon, O. and X. Liu. *DUDE: a Dialogue and Understanding Development Environment, mapping Business Process Models to Information State Update dialogue systems*. in *EACL 2006.*
[32] Larsson, S., R. Cooper, and S. Ericsson. *Menu2dialog*. in *IJCAI Workshop on Knowledge And Reasoning In Practical Dialogue Systems, 2001*.
[33] Omojokun, O. and P. Dewan. *Experiments with Mobile Computing Middleware for Deploying Appliance UIs.* in *In Proceedings of the 23rd International Conference on Distributed Computing Systems – Workshops*. 2003.
[34] Omojokun, O., *Interacting with Networked Devices*, in *Computer Sciences*. 2006, University of North Carolina.
[35] *Nevo for PDAs*.
[36] Sutton, J. and R. Sprague, *A Study of Display Generation and Management in Interactive Business Applications Tech. Rept. RJ2392(#31804)*. November 1978: IBM San Jose Research Laboratory.
[37] Myers, B., *User Interface Software Tools.* ACM Transactions on Computer-Human Interaction, March 1995. **2**(1): p. 64-103.
[38] Gajos, K., et al. *Fast And Robust Interface Generation for Ubiquitous Applications*. in *Proceedings of the Seventh International Conference on Ubiquitous Computing (UBICOMP'05)*.
[39] Sommerville, I., *Software Engineering Environments.* (IEEE Computing Series Number 7, CM007): p. ISBN 0 86341 077 4, 1986.
[40] Shneiderman, B., *CHAPTER 10 - "Response Time and Display Rate"*, in *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. 1998, Addison-Wesley Longman. p. 352-369.