

INTERACTING WITH NETWORKED DEVICES

Olufisayo Omojokun

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill
in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the
Department of Computer Science.

Chapel Hill

2006

Approved by

Advisor: Prasun Dewan

Reader: Charles Isbell

Reader: Ketan Mayer-Patel

Reader: Maria Papadopouli

Reader: Amin Vahdat

Reader: David Stotts

Reader: Richard Han

© 2006
Olufisayo Omojokun
ALL RIGHTS RESERVED

ABSTRACT

OLUFISAYO OMOJOKUN: Interacting with Networked Devices

(Under the direction of Prasun Dewan)

Networking technology has become applicable in domains beyond the conventional computer. One such domain currently receiving a significant amount of research attention is networking arbitrary devices such as TVs, refrigerators, and sensors. In this dissertation, we focus on the following question: how does an infrastructure deploy a user-interface for a single device or a composition of several ones?

We identify and evaluate several deployment approaches. The evaluation shows the approach of automatically generating device user-interfaces ‘on the fly’ as particularly promising since it offers low programming/maintenance costs and high reliability. The approach, however, has the important limitation of taking a long time to create a user-interface. It is our thesis that it is possible to overcome this limitation and build graphical and speech user-interface generators with deployment times that are as low as the inherently fastest approach of locally loading predefined code. Our approach is based on user-interface retargeting and history-based generation. User-interface retargeting involves dynamically mapping a previously generated user-interface of one device to another (target) device that can share the user-interface. History-based generation predicts and presents just the content a user needs in a device’s user-interface based on the user’s past behavior. By filtering out unneeded content from a screen, it our thesis that history-based generation can also be used to address the issue of limited screen space on mobile computers.

The above ideas apply to both single and multiple device user-interfaces. The multi-device case also raises the additional issue of how devices are composed. Current infrastructures for composing devices are unsuccessful in simultaneously providing high-

level and flexible support of all existing composition semantics. It is our thesis that it is possible to build an infrastructure that: (1) includes the semantics of existing high-level infrastructures and (2) provides higher-level support than all other infrastructures that can support all of these semantics. Such an infrastructure requires a composition framework that is both data and operation oriented. Our approach is based on the idea of pattern-based composition, which uses programming patterns to extract data and operation information from device objects. This idea is used to implement several abstract algorithms covering the specific semantics of existing systems.

ACKNOWLEDGEMENTS

I am deeply grateful to several people who have helped me throughout my years as a graduate student. First is my advisor, Professor Prasun Dewan, whose guidance, support, and encouragement allowed me to complete my thesis. I also want to thank the other members of my thesis committee, Professor Charles Isbell, Professor Maria Papadopouli, Professor Ketan Mayer-Patel, Professor David Stotts, Professor Richard Han, and Professor Amin Vahdat for their valuable comments and suggestions.

I would like to thank my parents for their loving support during all my years in and before graduate school. I additionally recognize my two brothers and two sisters for their emotional support during stressful times.

Finally, I gratefully acknowledge Microsoft and the National Science Foundation for the financial support provided by the following grants: ANI 0229998, EIA 03-03590 and IIS 0312328.

TABLE OF CONTENTS

	Page
Chapter 1: Introduction	1
Benefits of Deploying Software-based User-Interfaces	3
1.2 Deploying Single Device User-Interfaces	8
1.3 Deploying Multi-Device User-Interfaces	11
1.4 Thesis	14
1.5 Summary	14
Chapter 2: Related Work	16
2.1 Deploying Single Device User-Interfaces	16
2.1.1 Palm/Pocket-PC IR Control Programs	18
2.1.2 Jini (Service UI Approach)	19
2.1.3 MOCA.....	20
2.1.4 Cooltown.....	20
2.1.5 Universal Plug and Play (UPnP).....	20
2.1.6 ObjectEditor	21
2.1.7 Hodes' System	22
2.1.8 Personal Universal Controller (PUC).....	23
2.1.9 ICrafter	24
2.2 Deploying Multi-Device User-Interfaces	25
2.2.1 Cougar and TinyDB	26
2.2.2 Hodes' System	27
2.2.3 Palm/Pocket-PC IR Programs.....	28

2.2.4 WebSplitter	29
2.2.6 Speakeasy.....	34
Chapter 3: Analysis of Various Approaches.....	36
3.1 Overview of Metrics and Setup	36
3.2 User-Interface Flexibility.....	39
3.3 Programming Costs.....	46
3.4 Maintenance Costs.....	49
3.4.1 Predefined vs. Generation.....	49
3.4.2 Client-Factory and Third-Party Factories vs. Other Approaches	49
3.5 Efficiency	50
3.5.1 Space Costs	51
3.5.2 Deployment Time Costs	52
3.5.3 Operation Invocation Time Costs	63
3.6 Device Binding Time.....	64
3.7 Deployment Reliability.....	64
3.8 Conclusion	65
Chapter 4: User-Interface Retargeting.....	69
4.1 Overview.....	70
4.2 GUI Retargeting.....	74
4.3 SUI Retargeting	91
4.4 Evaluation	92
4.4.1 Source User-interface Selection Performance	95
4.4.2 Approach Selection Performance	97
4.4.3 Retargeting Performance	98
4.5 Conclusion	110

Chapter 5: History-based Generation.....	113
5.1 Approach.....	114
5.2 Evaluation	117
5.2.1 Generation Time Efficiency.....	118
5.2.2 Screen Space Efficiency	123
5.3 Conclusion	124
Chapter 6: Pattern-based Composition	125
6.1 Overview.....	126
6.2 Algorithms and Evaluation	136
6.2.1 ‘GUI Stack’ Composer	137
6.2.2 ‘GUI Merge’ Composer.....	138
6.2.3 ‘Do Sequence’ Composer	140
6.2.4 ‘Do All’ Composer.....	142
6.2.5 Query Composer.....	144
6.2.6 ‘Data Transfer’ Composer.....	146
6.2.6 ‘Conditional Connect’ Composer	149
6.3 Conclusion	152
Chapter 7: User-Based Composition	153
7.1 ML Approach.....	155
7.2 Experiments	156
7.3 Evaluation	158
7.3.1 Completeness	159
7.3.2 Task-based Grouping.....	160
7.3.3 ‘Do Sequence’ Discovery	162
7.4 Conclusion	163

Chapter 8: Conclusions and Future Work.....	164
Appendix A: Snapshots of Predefined and Generated GUIs	169
References.....	180

LIST OF TABLES

Table 1. An example composer registry.....	31
Table 2. An example composer registry.....	31
Table 3. Number of lines of user-interface code used for each device	48
Table 4. Amount of space consumed by the code of each device' handcrafted user-interface.....	52
Table 5. Retargeting flexibility – Hodes' System vs. Our Goals.....	74
Table 6. A summary of our 11 participants.....	94
Table 7. An evaluation of Tret's ability to predict the fastest command-only user-interface to retarget. { * The DVD player also serves as a music CD player}	97
Table 8. An evaluation of Tret's ability to predict the fastest command-and-state based user-interface to retarget. { * The DVD player also serves as a music CD player} ..	97
Table 9. For command-only UI deployment, a comparison of the approach predicted to be the fastest to the approach that actually measures to be the fastest.....	98
Table 10. For command-and-state based UI deployment, a comparison of the approach predicted to be the fastest to the approach that actually measures to be the fastest....	98
Table 11. Number of screens consumed by each device's command-only GUI.	114
Table 12. A summary of command filtering amounts for each device.....	120
Table 13. Number of commands required in each participant's set of history-based user-interfaces.	122
Table 14. The number of Ipaq screens required for full and history-based GUI.	123
Table 15. A classification of existing systems	130
Table 16. A count of each participants missed buttons.....	159
Table 17. A count of user-interface switches required for participants' common tasks.	161

LIST OF FIGURES

Figure 1. (a) The author controlling a TV, VCR, and projector; (b) An adhoc security-system composition consisting of a motion sensor and stereo.....	3
Figure 2. (a) Left, a VCR's on-board controls; (b) Right, a Traditional IR Remote.	4
The mobile computer approach, illustrated above, offers several additional benefits:	4
Figure 3. Two possible approaches to UI deployment: (a) deploying a UI from pre-installed code and (b) generating a UI.	9
Figure 4. A generated receiver user-interface on an Ipaq.	10
Figure 5. A depiction of cell phone (Motorola i710) and Pocket PC (Compaq Ipaq) screen size differences. The cell phone's screen is less than half of the Ipaq's.....	11
Figure 6. A conceptual view of how a pattern-based composer could query several sensors.	13
Figure 7. The general architecture abstracts existing infrastructures for deploying for single device UIs	16
Figure 8. Current UI deployment approaches.....	17
Figure 9. The IR port on the front end of the Ipaq for transferring data and controlling IR devices.....	18
Figure 10. A TV user-interface created using OmniRemote[2].....	19
Figure 11. A CD player HTML-based user-interface inside a Netscape Browser.	20
Figure 12. A depiction of how ObjectEditor works.....	22
Figure 13. A lamp UI generated by Hodes' System [15, 16].....	23
Figure 14. A portion of a stereo system user-interface generated by PUC [27]. Notice the cassette navigation buttons are specifically grouped together.	24
Figure 15. (left) the HTML generated for setting a projector input; (right) the rendered web page[28].	25
Figure 16. Two possible ways to organize composers.....	26
Figure 17. A compound UI for a set of lights.	28
Figure 18. A Websplitter presentation in which audio is sent to a stereo and frames are shown in display devices [13].	29

Figure 19. A lights composition.....	30
Figure 20. An example composer registry.....	31
Figure 21. (left) a <i>Celadon PIC Link</i> IR module (right) an <i>X10 FireCracker CM17A</i> module.....	37
Figure 22. Sample programming interfaces (for the receiver and lamp).....	38
Figure 23. Command-only receiver GUIs written using Java Swing: (a) a predefined GUI that mimics the device's remote control and (b) a GUI generated fully automatically by ObjectEditor.	40
Figure 24. Command and state-based receiver GUIs: (a) predefined GUI and (b) generated by ObjectEditor fully automatically.	42
Figure 25. A depiction of our experimental SUI generator.	45
Figure 26. UI generation vs. the predefined approach.	47
Figure 27. The downloaded components of the factory and generation approaches.....	54
Figure 28. Command-only GUI deployment times for all six devices (using the laptop, ObjectEditor preloaded in memory, and a wired LAN connection).	56
Figure 29. Command-and-state based GUI deployment times for all six devices (using the laptop, ObjectEditor preloaded in memory, and a wired LAN connection).	57
Figure 30. Command-and-state based GUI deployment times for the projector, lamp, and receiver (using the Ipaq and a wired LAN connection).	58
Figure 31. A visual display of the significant differences between Ipaq and laptop GUI deployment.....	59
Figure 32. Command and state based receiver GUI deployment times using the laptop and different network speeds.	60
Figure 33. Command-only SUI deployment times for the projector, lamp, and receiver (using the Laptop and a wired LAN connection).....	62
Figure 34. A graphical representation of the unique benefit(s) of each approach.....	68
Figure 35. Retargeting a UI between two lights on different floors.	69
Figure 36. Retargeting between two lights in different rooms under Hodes' System.	70
Figure 37. Levels of source UI flexibility.....	71

Figure 38. User-interfaces requiring different levels of source user-interface flexibility.	71
Figure 39. Devices with identical programming interfaces share identical user- interfaces.	72
Figure 40. Retargeting between a dimmable and non-dimmable lamp.	73
Figure 41. Two different VCR programming interfaces that can share the same UI.	73
Figure 42. The maximum N value for <i>Tadd_btn</i> comes from retargeting a light UI to a receiver.	81
Figure 43. A depiction of part of our profiling experiments: (a) finding the time it takes to add 4 buttons to an empty UI, (b) finding the time it takes to remove 4 buttons from a UI, and (c) finding the time it takes to remap 4 buttons on a UI.	82
Figure 44. The time it takes to add new buttons to an empty GUI (slow vs. fast laptop).	83
Figure 45. The time it takes to add new property widgets to an empty GUI (slow vs. fast laptop) The dashed lines correspond to the slow laptop and the bold lines correspond to the fast laptop.	83
Figure 46. The time it takes to remove pre-existing property widgets on a GUI (slow vs. fast laptop).	84
Figure 47. The time it takes to remap pre-existing property widgets on a UI (slow vs. fast laptop)	84
Figure 48. A graph illustrating time differences between the three clients. It also shows that widgets representing different types can yield different operation times— particularly on the Ipaq.	85
Figure 49. The differences between adding, removing, and remapping a property widget.	86
Figure 50. Our tool for automatically recording device interactions at a person’s home.	92
Figure 51. A close up of the IRman serial port device.	93
Figure 52. The conference room we used.	95
Figure 53. Homogeneous retargeting of command-and-state GUIs using the fast laptop and wired LAN connection.	100

Figure 54. Homogeneous retargeting of command-only SUIs using the fast laptop and wired LAN connection.	101
Figure 55. A graph comparing the homogeneous retargeting times the to the corresponding times of competing approaches (using the Ipaq and a 100Mbps connection).	102
Figure 56. Heterogeneous retargeting of command-only GUIs vs. competing approaches (using the fast laptop and a wired LAN connection).	104
Figure 57. Retargeting times of command-and-state based GUIs vs. the corresponding times of competing approaches (using the fast laptop and a wired LAN connection).	104
Figure 58. Retargeting times of command-and-state based GUIs vs. the corresponding times of competing approaches (using the Ipaq and a wired LAN connection).	105
Figure 59. Retargeting times of the receiver command-and-state based GUI vs. the corresponding times of competing approaches (using the fast laptop and dialup connection).	106
Figure 60. Cache-based retargeting times of the command-only GUIs vs. the corresponding times of competing approaches (using the fast laptop and wired LAN).	107
Figure 61. Cache-based retargeting times of the command-and-state based GUIs vs. the corresponding times of competing approaches (using the fast laptop and wired LAN).	108
Figure 62. Cache-based retargeting times of the command-and-state based GUIs vs. the corresponding times of competing approaches (using the Ipaq and wired LAN).	108
Figure 63. Cache-based retargeting times of the receiver command-and-state based GUI vs. the corresponding times of competing approaches (using the fast laptop and dialup connection).	110
Figure 64. An entire receiver GUI (left) vs. a receiver GUI containing all of the commands the owner (author) typically needs (right).	113
Figure 65. Number of usage days required for the participants to complete their common tasks on their respective devices.	116
Figure 66. History-based GUI generation performance using the laptop and wired LAN connection.	119
Figure 67. History-based GUI generation performance using the Ipaq and wired LAN connection.	119

Figure 68. History-based SUI generation performance using the laptop and wired LAN connection.	121
Figure 69. The receiver’s history-based GUI on a single Ipaq screen. With the available space from filtering buttons, the remaining buttons can be stretched to fill the screen.	124
Figure 70. Extracting the state and operations from a lamp’s programming interface..	136
Figure 71. A stacked GUI for watching movies—based on the author’s TV, DVD player, and receiver.	138
Figure 72. A GUI for selecting desired buttons for a target task.	140
Figure 73. A GUI for creating a ‘watch a DVD’ button.	142
Figure 74. A ‘do all’ GUI for a set of lamps.	143
Figure 75. An example GUI for querying rainfall sensors with several attributes.	145
Figure 76. A data transfer GUI for cameras and display devices.	149
Figure 77. A ‘conditional connect’ GUI for creating the adhoc lamps and motion detector security system.	151
Figure 78. Our setup containing the participants’ remote, several blank sheets (screens), and button squares.	157
Figure 79. The three user-interfaces that P5 created.	158
Figure 80. A projection of P6’s clusters	161
Figure 81. A projection of P2’s clusters	168

Chapter 1: Introduction

Networking technology has become applicable in domains beyond the conventional computer. One such domain currently receiving a significant amount of research attention is networking arbitrary devices such as TVs, refrigerators, and sensors. There are a number of compelling reasons that make this idea desirable.

One reason is that with a network connection, a device can enhance some general functionality that it already provides. To illustrate, today's DVD players are capable of showing extra feature content about a movie. Such content is 'burned' on a DVD when it is released. With a network connection, a DVD player could possibly connect to a movie producer's server and download additional content that consists of interesting information that develops after a DVD is released. This ability could be particularly useful in the case of documentaries, which contain facts and data that may become outdated.

Another related reason for networking a device is to allow it to offer totally new functionality that would otherwise be impossible. Consider the following example that has captured the imagination of many: a refrigerator with built-in sensors that could allow it to discover when certain important food items are nearly finished or expired. With a network connection, the refrigerator could notify its owner, who is away from home, to purchase new food. Even more complex, it could connect to the server of the nearest grocery store and order new food. The owner could simply pick the order up and avoid actual shopping [23].

Yet another reason for networking devices is to remotely log the interactions that users have with them. Researchers at the University of Arizona are building a framework in which device manufacturers can make use such a facility [35]. In this framework, devices execute software agents that record and send certain user-initiated events to servers owned by their manufacturers. With such information, manufacturers can gain a clear understanding of how consumers use their devices—thus leading to possible

design improvements. The networked refrigerator from earlier could, for example, notify its manufacturer of the number of times its doors are opened. Its manufacturer could use this information when designing durable door hinges for future refrigerators. Other parties beyond device manufacturers might find it useful to access logs of user-device interaction. For example, advertisers may want to use logs from remotely loggable TV sets to learn the typical program watching habits of a population of people. Given such information, they can predict the most opportunistic times to place ads and later find an approximate number of likely viewers.

Networking devices can also allow for new and different ways for users to interact/control them. For example, it can allow users to interact with devices using software-based user-interfaces deployed on mobile computers. Figure 1a demonstrates such a case by showing the author using an HP680 Jornada handheld computer to interact with a networked TV, VCR, and projector in a classroom.

It is possible to create software-based user-interfaces for single devices and for combinations of them. A single-device user-interface allows users to control and possibly view the state of a single networked device. For example, it could allow a person driving home on a hot day to use a cell phone to set the thermostat level of the house's air-conditioning system so that the temperature is cool before getting there. It could also allow the driver to set a TiVo box at home to record an upcoming TV show if there is heavy traffic on the road.

A multi-device user-interface, on the other hand, allows users to dynamically compose the services offered by multiple networked devices. For example, it could allow a security guard to compose a group of lights in an office building's hallway so that they can be automatically powered on and off by using a single control (e.g. an 'all lights power' button). In addition, electronic locks on the possibly many exit doors in the building could be composed together so that the security guard could automatically activate and deactivate them by issuing a single command. This feature would be highly desirable in the event of a building fire. These examples demonstrate one of the reasons for composing a group of devices together: to provide a more efficient means to

completing a given task involving the devices than what is provided by using their individual controls. Another reason is to form a composite unit that provides functionality that the individual devices cannot achieve separately. A stereo and several motion detectors in a house, for example, could be composed together to form an ad hoc security system (Figure 1b). When a detector senses motion, it triggers the stereo to blast music. The same motion detectors could coordinate with various cameras around the house so that any sensed motion triggers the nearest camera to snap pictures of a possible intruder. As the person moves around, other cameras are triggered.

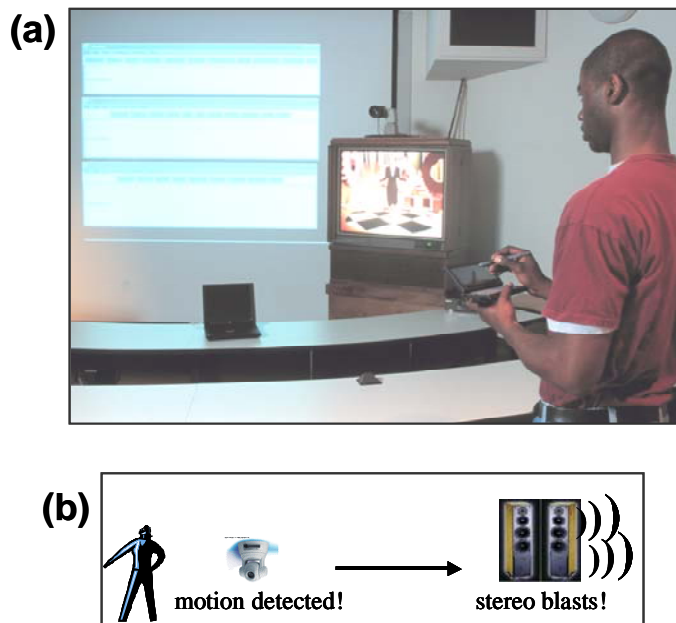


Figure 1. (a) The author controlling a TV, VCR, and projector; (b) An ad hoc security-system composition consisting of a motion sensor and stereo.

In this dissertation, we are particularly interested in the ability to interact with individual devices and their compositions by using software-based user-interfaces deployed on mobile computers. This ability, itself, has many benefits.

Benefits of Deploying Software-based User-Interfaces

Today, it is possible to interact with devices by using hardware-based user-interfaces. Examples of such user-interfaces are on-board and traditional remote controls. On-board controls require users to be within arms reach of the devices they wish to control (Figure 2a). However, it may not always be possible to reach a device. For instance, a presenter

may not be able to reach a projector mounted on a high ceiling. Traditional infrared (IR) and X10 remote controls address this limitation by allowing users to control devices from afar (Figure 2b).

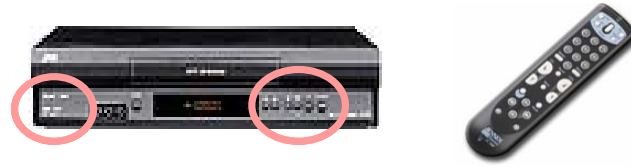


Figure 2. (a) Left, a VCR's on-board controls; (b) Right, a Traditional IR Remote.

The mobile computer approach, illustrated above, offers several additional benefits:

- *More 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. A mobile computer would be a more universal control than a traditional remote control, for several reasons:
 - *Arbitrary number of device instances:* 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, do not incur such restrictions. Therefore, they 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. This approach could also allow them to use a user-interface that composes lights in one building to control light compositions in other buildings.
 - *Control of dissimilar device types:* 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 [3]. A mobile computer can serve as a single control for arbitrarily different kinds of devices.

- *Automatic late binding to devices:* Traditional remote controls (e.g., those that are not universal) support early binding. As result, they are bound to specific device instances when they are built. Late binding allows a remote control to bind to different device instances after it is built. Universal remote controls support late binding. However, they require users to manually enter appropriate 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 it 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 [4, 8, 12, 20, 36].
- *More 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 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 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.
- *More control:* Perhaps a more intriguing 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 physical user-interfaces

offered by traditional controls [27]. For example, mobile computers can offer the following kinds of enhancements:

- *View device output:* Unlike a conventional remote control, a mobile computer is an output device. It can thus display application output such as car diagnostic readings and water sprinkler settings. The ability to display output on a remote control may not seem important if the output can also be displayed on a device connected to it, such as a VCR displaying output on a connected TV. However, there are at least two situations under which this feature is useful. First, the output device may be used to display other information of interest. For example, a TV may be showing an interesting program while VCR settings are being entered and displayed on the mobile computer. This approach avoids consuming the TV screen so that a viewer can watch programs. Second, and more important, the device data sometimes needs to be viewed when the mobile computer is no longer within sight or connected to the output device. For example, TV data may be viewed when parents are at work and no longer at home to check what their kids are watching.
- *Offline editing and synchronization:* Device data can also be edited in the offline mode, and later synchronized. For example, a person can edit a TiVo's program record settings in the offline mode and then later synchronize them. This facility has been found to be useful in some traditional computer-based applications such as address books and, as the example shows, it can also be useful for device interaction.
- *Personalization:* Mobile computers can create device user-interfaces that are tailored to a specific user's habits and information needs. For example, they can create user-interfaces that automatically feed user-specific data to shared devices such as favorite channels and volume levels to TVs, PINs to ATM machines, credit card numbers to a coke machines, preferred car-seat tilt angle to cars, and files to printers. A mobile computer could record data such as

PINs and credit card numbers during a user's first interaction with a device. It could then automatically enter such data in later interactions.

The above benefits apply to using software-based user-interfaces to interact with both single devices and their compositions. Using a software-based approach to compose devices has certain additional benefits:

- The hardware-based composition approach requires hardwiring devices together. This is not easy since it requires special experience in electronics. A software-based approach can offer high-level user-interfaces for easily composing devices.
- Because it requires wiring for every combination of devices that the system can compose, the hardware approach does not scale over distance. In the hardware-based approach, providing the ability to turn off all hallway lights in a building requires manually wiring them to a master switch. This task could require extensive wiring if the building has many floors and there are several master switches. The software-based approach scales better over distance because it can use the Internet. Also, the lights could take advantage of a wireless network available in the building—thus offering a ‘plug and play’ like functionality.
- For proprietary and warranty reasons, device manufacturers may not even allow end-users to examine and change the hardware makeup of their devices. This limits the composition flexibility of the hardware-based approach. To allow flexibility and keep the hardware designs of their devices private, device manufacturers can provide a means to compose their devices using software.

The reasons above are only proposed benefits of using mobile computers to interact with devices. Determining which of these reasons are actually useful requires building infrastructures and experimenting with users. Today, several such infrastructures have been built, which include: Palm/Pocket-PC IR programs [1, 2], HP's Cooltown [14], IBM's Moca and Websplitter [13], Microsoft's Universal Plug and Play (UPnP) [20], Sun's Jini [36], CMU's Personal Universal Controller (PUC) [26, 27], Hodes' System [15, 16], Cornell's Cougar [5], Berkeley's TinyDB [22], Stanford's ICrafter [28], and

PARC/Georgia Tech’s Speakeasy (also called Obje)[9]. Building these infrastructures entails addressing several complex and diverse issues. An example issue is how a mobile computer discovers the available devices within a network or physical space. Another issue is security, which is how an infrastructure prevents non-privileged users from invoking commands on the devices it contains. In this dissertation, we focus on the user-interface deployment issue: how does an infrastructure deploy a user-interface for a single device or a composition of several ones? In particular, we address several limitations of current approaches to this issue in the single-device and multi-device cases.

1.2 Deploying Single Device User-Interfaces

Existing infrastructures for deploying single device user-interfaces demonstrate diverse approaches to addressing this issue. These approaches have striking differences. One approach involves executing preinstalled (device specific) user-interface code on a client’s local storage (Figure 3a). Imagine if the vacationer mentioned earlier used this approach. Sometime before leaving home, this person would preinstall a user-interface program for specifically controlling the security system on the mobile computer. The security system’s manufacturer could have provided this program to its customers. Another approach involves a client dynamically creating a user-interface based on the functional description of a target device (Figure 3b). With this approach, the vacationer does not need to pre-install any user-interface code that is specific to the security system or any other device that will later be of interest. The mobile computer simply needs be able to access a possibly local user-interface generator. On the other hand, intuitively, it should offer relatively long deployment times because it involves creating a user-interface ‘on the fly’—especially when compared to directly loading handcrafted code from disk. Later, we will show that this intuition is valid. In fact, generation times are actually much longer (by multiples) than the corresponding deployment times of all other approaches.

The above mentions just two of the several existing approaches that we will describe in this dissertation. Still, it is enough to imply that an approach can have certain significant advantages over another. Understanding such advantages and disadvantages is important when building an infrastructure for interacting with networked devices. However,

current approaches have not been previously compared in a systematic manner. Therefore, their specific strengths and weaknesses are not well known. Based on the notion that striking differences exist among them, it is our first hypothesis that each approach offers a set of unique benefits. The benefits of an approach would therefore provide a reason for why it exists.

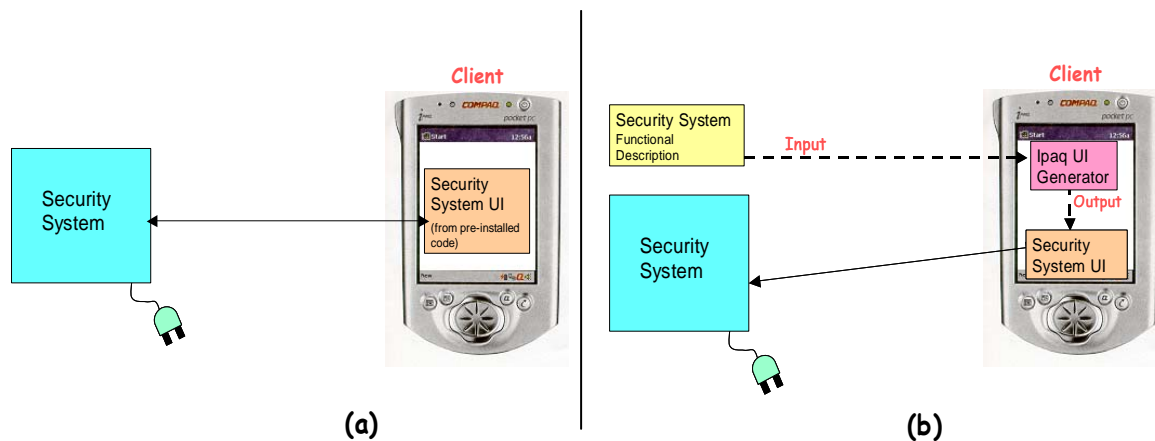


Figure 3. Two possible approaches to UI deployment: (a) deploying a UI from pre-installed code and (b) generating a UI.

Given a reason for using each approach, attempting to address the limitations of each is thus worthwhile. Most approaches, however, do not offer a means to feasibly address their limitations. Consider the approach of locally loading pre-installed user-interface code. If a user wishes to interact with a device for which there is no pre-installed user-interface code, the approach fails. Such failures cannot be avoided in ad hoc and unforeseen interactions. The generation approach, on the other hand, can support such interactions. However, recall that this approach has the limitation of long deployment time. It is our second hypothesis that it is possible for GUI and SUI generators to have deployment times that are often as good as or noticeably better than the inherently fastest approach of locally loading device-specific user-interface code. One idea for achieving such competitive generation times is user-interface retargeting. It involves dynamically mapping a previously generated user-interface of a (source) device to another (target) device that can share the user-interface. By recycling parts of a previously generated user-interface of a device that a user is not using, we show that a generator can significantly speed up the creation of a user-interface.

Another idea for supporting time-efficient generation is lazy generation, which involves opportunistically generating user-interfaces that consist of subsets (rather than all) of the functionally provided by their corresponding devices. It supports the principle that the less content a user-interface will contain, the less time it should take to generate the user-interface. Within the scope of lazy generation, we focus on generating history-based user-interfaces. Such user-interfaces are generated to present only the commands a user typically uses (or needs) from a device, based on the user's past behavior with the device. Hence, the assumption is that the content a user needs is generally less than the content needed in presenting the device's entire capabilities.

This assumption implies that history-based generation could also be used to address the problem of limited screen space offered by mobile computers when displaying GUIs. To illustrate this problem, consider an A/V receiver user-interface created by a user-interface generator built here at UNC. It only consumes one screen on a laptop. However, it spans three screens on the Ipaq (Figure 4). Imagine the user-interface for cell phones, which generally have screen sizes that are fractions of the size of Ipaq's (Figure 5).

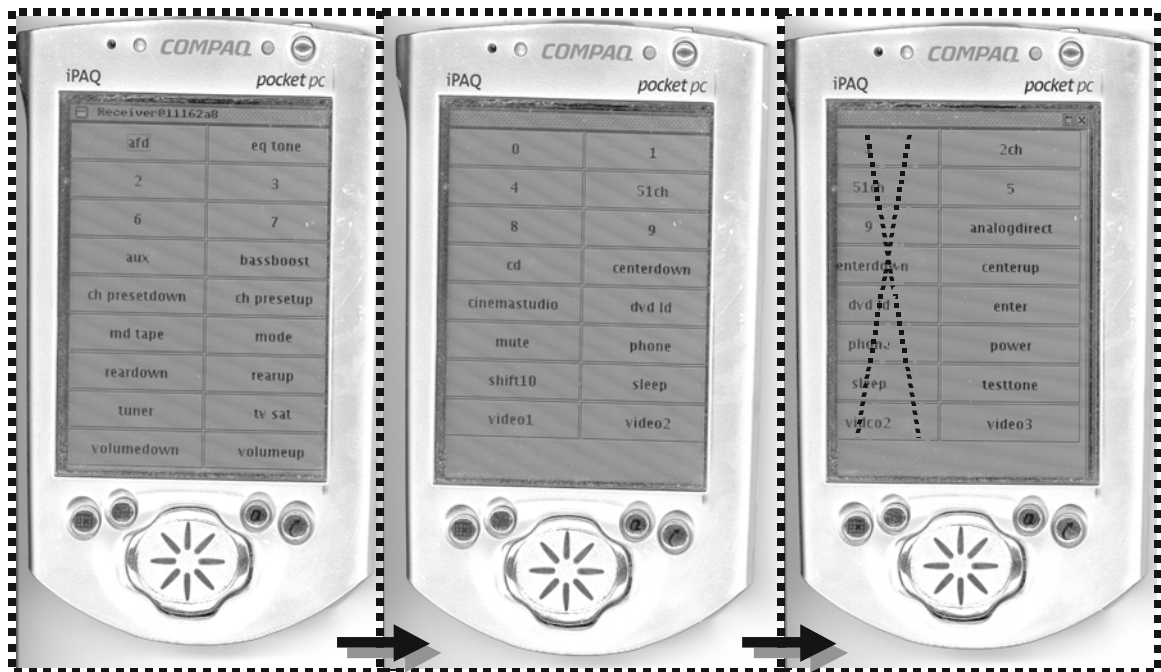


Figure 4. A generated receiver user-interface on an Ipaq.

In general, this problem forces users to tediously search within user-interfaces by scrolling and tabbing through several screens in order to control a device. It is our third hypothesis that history-based user-interfaces can consume significantly fewer screens than their corresponding full device user-interfaces.



Figure 5. A depiction of cell phone (Motorola i710) and Pocket PC (Compaq Ipaq) screen size differences. The cell phone's screen is less than half of the Ipaq's.

1.3 Deploying Multi-Device User-Interfaces

As mentioned earlier, we also address existing limitations of infrastructures for deploying software-based multi-device user-interfaces. Such infrastructures must additionally offer users with a means to composing devices. Existing examples demonstrate different approaches to supporting such functionality.

Some infrastructures provide users with already programmed mechanisms for achieving desired compositions. For example, Cougar and TinyDB are two infrastructures that provide mechanisms for querying a network of sensors. They can support scenarios such as a person querying presence sensors in the rooms of an office building to find a free place to work. This person executes a single command to find rooms with no human presence rather than requesting the information individually from each of a possibly large set of sensors. The two infrastructures are relatively high-level because they: (a) provide a query language for users and (b) automatically perform queries and return results. However, they do not flexibly support composition. Neither of them supports any of the non-query-based kinds of composition semantics illustrated thus far. For example, neither provides mechanisms for composing a sensor with a stereo

to form the ad hoc security system we mentioned earlier. Our summary of existing systems (Chapter 2) will show that, in fact, all existing high-level infrastructures share this general problem of limited composition flexibility. In particular, each high-level infrastructure supports composition semantics that no other high-level infrastructure supports.

Infrastructures have been built for generically supporting composition. These infrastructures, however, are low-level since they place much of the programming burden on users or end-programmers of these infrastructures. Our later discussion of existing systems will also show that this burden is not small largely due to the combinatorics involved in flexibly supporting composition. Just the few examples in this chapter imply that there are *many* different ways that a device can be dynamically composed with *many* other devices of *arbitrary* kinds. Also, these devices can be composed based on their possibly *many* operations (to simultaneously invoke shared operations, for example) and/or data entities (to perform queries, for example).

Based on the above discussion, it seems that existing approaches to composing devices must tradeoff high level support for composition flexibility. Specifically:

- 1) each existing high-level infrastructure supports composition semantics that no other high-level infrastructure supports
- 2) each low-level infrastructure can flexibly support each of the existing composition semantics but has the programming cost of writing composer mechanisms.

It is our fourth hypothesis that a new infrastructure can be built to address this problem by meeting the two conditions below:

- 1) supports the composition semantics of existing high-level infrastructures.
- 2) provides higher-level support than all other infrastructures that can support all of these semantics.

Our approach is based on the use of programming patterns [29] when coding device objects. Programming patterns are rules for defining the names, parameter types, and

return types of an object's public methods for the purposes of exposing the object's structure and semantics to external software tools. The Java Beans framework demonstrates this idea by allowing a programmer to describe the state properties of an object in the object's programming interface. To export a property named `<Property Name>` of type `<Property Type>`, programmers must implement methods with the following constraints:

- 1) `public <Property Type> get<Property Name>()`
- 2) `public void set<Property Name>(<Property Type>)`

Sensor programmers could, for example, implement the following methods to export a state property named 'motion detected' that is a `boolean` type:

- 1) `public boolean getMotionDetected()`
- 2) `public void setMotionDetected(boolean)`

A sensor's `getMotionDetected()` method returns `true` if motion is detected. It returns `false` if no motion is detected. Intuitively, a query-based composer could be built that extracts the 'motion detected' status from sensors offering this method to discover, for example, whether there is a free place for someone to do work (Figure 6).

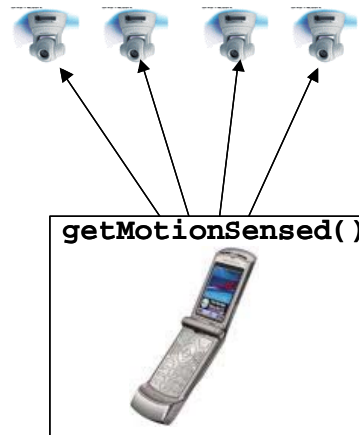


Figure 6. A conceptual view of how a pattern-based composer could query several sensors.

We specifically hypothesize that programming patterns can be used to allow us to write high-level composer mechanisms that automatically extract the necessary information from device objects for supporting all existing composition semantics.

1.4 Thesis

It is our thesis that it is possible to overcome the several limitations presented in this chapter. In particular, our thesis verifies the following hypothesis:

- I. *Uniqueness Hypothesis*: Each existing user-interface deployment approach offers a unique benefit, thus providing a reason why each exists.
- II. *Time-Efficient Generation Hypothesis*: It is possible for SUI and GUI generators to use retargeting and history-based generation to offer deployment times that are often as good as or noticeably better than the inherently fastest approach of locally loading device-specific user-interface code.
- III. *Screen-Space-Efficient Generation Hypothesis*: History-based generation can also be used to create user-interfaces that consume significantly fewer screens than their corresponding full device user-interfaces
- IV. *High-level and Flexible Composition Hypothesis*: It is possible to build a composition infrastructure, based on programming patterns, that is simultaneously more high-level and flexible than the state of the art.

1.5 Summary

In this chapter, we introduced several reasons for networking devices. One reason, which is our primary focus, is to allow users to interact with single devices and their compositions by using software-based user-interfaces deployed on mobile computers. We discussed several benefits and limitations of the state of the art in this area. In addition, we presented several hypotheses for overcoming the described limitations. It is our thesis that all of our stated hypotheses are true. We will develop this thesis in the following chapters, which are organized as follows. Chapter 2 describes current approaches to deploying software-based user-interfaces and the existing systems that use them. This chapter will also present the earlier mentioned tradeoff between high-level support and composition flexibility as exhibited by existing multi-device based systems. In Chapter 3, we qualitatively and quantitatively evaluate various deployment approaches mentioned in Chapter 2. This evaluation subsequently leads to a proof of the Uniqueness

Hypothesis. Chapters 4-7 address the three latter hypotheses by respectively focusing on our three main ideas—retargeting, history-based generation, and pattern-based composition. Each chapter presents: (1) the design issues of its corresponding idea and our approaches to addressing them, (2) our implementation of the idea, and (3) an evaluation of how well the idea achieves its associated goal(s). Finally, Chapter 8 presents our conclusions and future work.

Chapter 2: Related Work

Our research is related to existing infrastructures for deploying software-based user-interfaces for single and multiple devices.

2.1 Deploying Single Device User-Interfaces

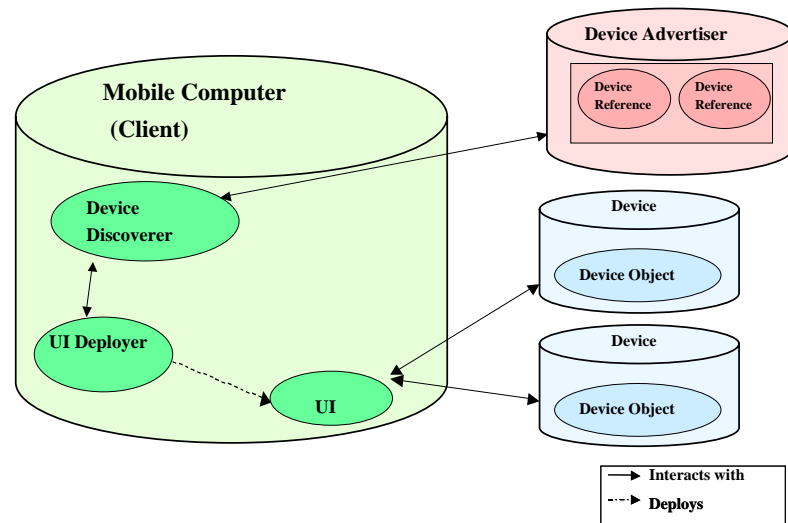


Figure 7. The general architecture abstracts existing infrastructures for deploying for single device UIs

Figure 7 shows a general architecture that abstracts existing infrastructures for deploying single device user-interfaces. The architecture consists of several components: mobile computers, devices, device objects, device advertisers, device references, device discoverers, user-interface deployers, composers, and user-interfaces. 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.

Given this general architecture, in the single-device case, we are concerned with the following question: how does a user-interface deployer produce an appropriate user-interface that can interact with the object of a user's target device? We separate current forms (Figure 8) of user-interface deployment into two high-level approaches: user-interface generation and the predefined approach. The predefined approach places pre-existing user-interface code at well-known servers for user-interface deployers to find and execute. Since these servers behave as factories [11] supplying user-interface code, they are called user-interface factories. Based on whether the location of the factory is the client or some other location, an approach is respectively classified as client-factory or remote-factory. A previous scenario from Section 1.2 illustrates the client-factory approach, in which the vacationer pre-installs the security system's user-interface on a mobile computer before leaving home. A remote-factory may be on a device or some third-party server. Under the device factory approach, the vacationer would download the user-interface code directly from the security system. Using the third-party factory approach, the vacationer could download code from a server at home or from the security system manufacturer's website.

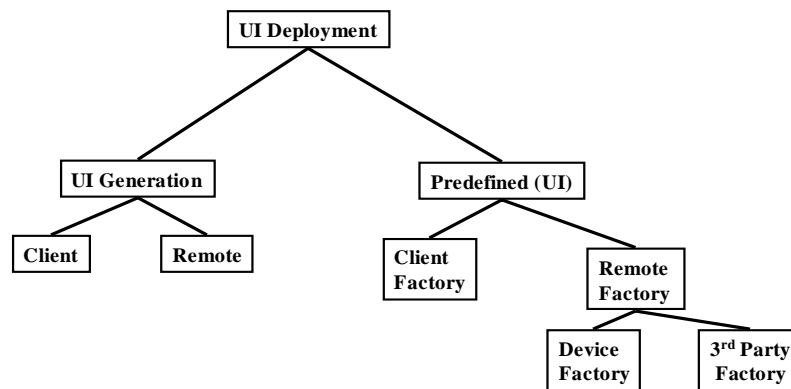


Figure 8. Current UI deployment approaches

Recall that the converse of the predefined approach is the user-interface generation approach because it does not require devices to be loaded with pre-defined user-interface code. Again, a user-interface generator dynamically creates an appropriate user-interface by using information extracted from a device's functional description. The generator can reside on the client device or on remote machine.

To show how all approaches could work, we will now summarize several commercial and research infrastructures that demonstrate them: Palm/Pocket-PC IR Control Programs, Jini, Moca, CoolTown, UPnP, ObjectEditor, Hodes' System, Personal Universal Controller, and ICrafter.

2.1.1 Palm/Pocket-PC IR Control Programs



Figure 9. The IR port on the front end of the Ipaq for transferring data and controlling IR devices.

Many of today's palmtop computers offer IR ports (Figure 9) that are typically used to transmit data between one another. Programs, such as OmniRemote[2] and Nevo[1], have been written for using these IR-ports to also control devices. In general, these programs provide users with user-interface building 'wizards' for creating and arranging buttons of a given device. During this process, users must also teach the system what IR-signals to emit for each button. Users can achieve this task in a manner that is similar to traditional remote controls. That is, they can enter predefined codes that are associated with specific devices. To allow users to create user-interfaces for devices that have no predefined codes, these programs also offer an IR recording feature. With this feature, users can push the buttons on the traditional remote controls of their unknown devices and record the signals emitted. They must then match the recorded signals to the corresponding buttons on the user-interfaces they created. After creating user-interfaces

(Figure 10), users can then save them on their palmtops for future use. Thus, Palm/Pocket-PC IR control programs support the client-factory approach.

A problem with IR control programs is that they require users to be in the vicinity of the devices they wish to control. Recall from earlier that this limitation is inherent of any IR-based method of device interaction. The infrastructures we discuss below avoid this limitation by supporting device interaction over the Internet.



Figure 10. A TV user-interface created using OmniRemote[2]

2.1.2 Jini (Service UI Approach)

Sun Microsystems created Jini[36] as a general infrastructure for building Java-based distributed systems. This infrastructure can also be used to network actual devices. It provides a framework that allows: (a) devices to join a network, (b) clients to discover devices, (c) and clients to access references (stubs) of remote devices to directly interact with a device (e.g. make remote procedure calls).

To deploy user-interfaces in Jini, Sun proposes the Service UI framework[39], which adopts the third-party factory approach. In this framework, a client accesses a factory on a machine with a well-known network location. It provides the factory with: (1) a description of the target device and (2) a description of itself. The remote-factory uses this information to ensure that it can provide code that presents a compatible user-interface for interacting with the target device. For instance, it could use a description of the client's screen size to ensure that it can provide code that presents a user-interface that fits properly. If such code is available, the client simply downloads and executes it.

A limitation of the Service UI framework is that it is based on Java. As a result, clients that cannot run a JVM are unable to interact with Jini-based devices.

2.1.3 MOCA

Similar to Jini, IBM's MOCA[4] is a Java-based infrastructure for building distributed systems, possibly containing networked devices. However, MOCA separates its user-interface deployment from its Java dependency. Devices can execute Java servlets that provide HTML-based user-interfaces to their clients (Figure 11). Thus, MOCA supports the device factory approach. Clients that support the HTML web standard can interact with a MOCA device.

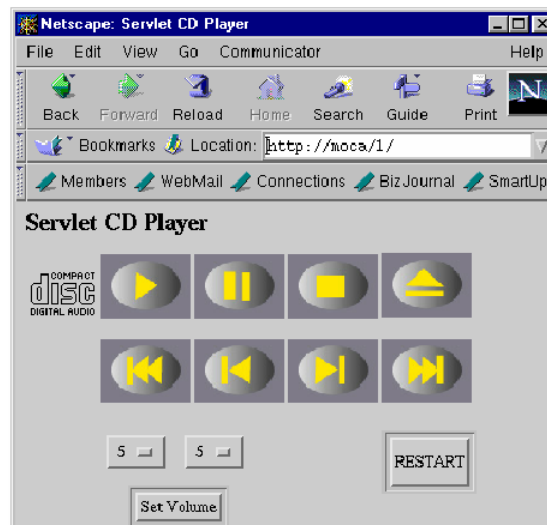


Figure 11. A CD player HTML-based user-interface inside a Netscape Browser.

2.1.4 Cooltown

HP's Cooltown[19] is another infrastructure that supports a web-based device factory approach. However, unlike MOCA, it does not require devices to be implemented using a specific language. It simply expects devices to execute webserver that provide HTML-based webpages that present user-interfaces.

2.1.5 Universal Plug and Play (UPnP)

Like Cooltown, Microsoft's UPnP[7] is designed to be fully language-neutral. UPnP devices also execute web-servers that provide HTML-based user-interfaces for clients to

download. The two infrastructures, however, significantly differ in how they address other areas such as discovery and security. To illustrate, UPnP supports the AutoIP protocol[38], which allows devices to dynamically join a network by assigning themselves an IP address. Cooltown devices, on the other hand, require an administrator to manually register them to a network. Further, Cooltown offers specific mechanisms to address security while UPnP currently does not.

2.1.6 ObjectEditor

ObjectEditor, developed here at UNC, is an example of the client-side generation approach. It is fairly complex system and has been used in the computer science department to generate user-interfaces for various research projects and also teaching. Here, we describe those aspects of it that specifically apply to generating device user-interfaces.

ObjectEditor can generate a GUI displaying the state properties and operations of a device coded as a Java object. It assumes that these components are described using programming patterns. In particular, state properties are described by signatures adhering to the Java Beans conventions mentioned earlier in Section 1.3. ObjectEditor supports additional kinds of conventions for describing state properties. However, these conventions are beyond our scope of device user-interface generation. Signatures that are not used to export state properties describe operations. To illustrate, the method signature `‘public void power()’` describes the ‘power’ operation for turning the TV on and off.

ObjectEditor creates a button and/or a menu item for each operation. It organizes these buttons and menu items in alphanumeric order on the user-interface. For each (possibly structured) property, the generator maps it to a (possibly structured) widget for displaying its value. The generator then initializes each widget with the result of the associated property’s getter method. In the sensor example in Section 1.3, the sensors ‘motion detected’ property could map to a checkbox for displaying its `boolean` values. If the sensor detects motion (i.e. `getMotionDetected()` returns `true`), the checkbox is checked, otherwise, it is unchecked.

To control a device, a user can select the menu items and push buttons on the user-interface or edit the values displayed by property widgets. Activating a button or menu item results in ObjectEditor invoking the associated method. If the method has parameters, the generator creates a dialog box consisting of widgets for entering desired parameter values. Suppose that a TV offers a `sleep(int)` method that accepts the number of minutes to wait before it automatically shuts off. If a user pushes the method's button, ObjectEditor would generate a dialog box providing a textbox for entering the sleep time.

When a user edits the value in a property widget, the generator invokes the setter method of the associated property, passing the new value as a parameter. For example, when a user types in a new TV channel in the channel property's textbox, ObjectEditor would invoke `setChannel()` with the new channel as a parameter. Figure 12 illustrates this entire process.

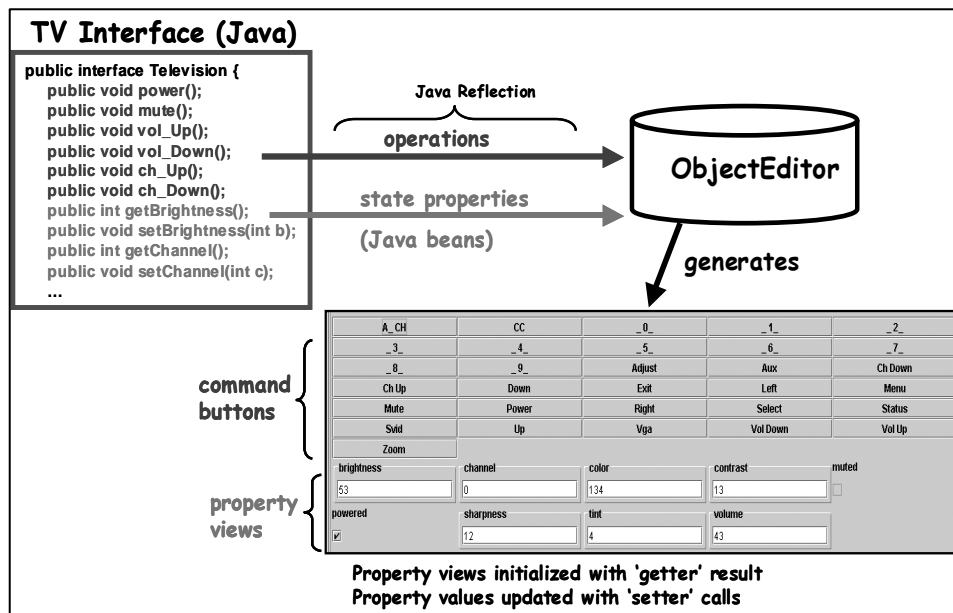


Figure 12. A depiction of how ObjectEditor works.

2.1.7 Hodes' System

ObjectEditor is language-dependent because it requires devices to be coded in their native language, Java. This limits the kinds of devices for which it can generate user-interfaces.

Hodes' System, a client-side generator, overcomes this limitation by offering a language-neutral generator. It generates user-interfaces from XML-based functional descriptions of devices (called services), such as the lamp description below:

```
<service name='lamp'>
  <label>lamp</label>
  <addrspec>sn140.cs.unc.edu/0001</addrspec>
  <method name='power'>
    <param lextype="enum:on,off,dim"> state </param>
  </method>
</service>
```

These descriptions consist of several tags for specifying values for the name, methods, method parameter types, and address of a service. Hodes' System generates a GUI that consists of a button for each method and an appropriate set of widgets for entering parameter values. The description above would be used to generate a user-interface that resembles the one shown in Figure 13. In the user-interface, the 'power' method parameter, called 'state', maps to an option box containing choices for each possible value. Once a user pushes a method's button, the generator performs a remote procedure call—sending the method's name and parameter values to the service's network address specified by the <addrspec> tags.



Figure 13. A lamp UI generated by Hodes' System [15, 16].

Although Hodes' System offers the flexibility of a language-neutral approach, it requires programmers to take the time to write descriptions in a separate language from the one in which their devices are coded.

2.1.8 Personal Universal Controller (PUC)

CMU's PUC system[27] also supports client-side generation of user-interfaces from XML-based device descriptions. PUC's device descriptions, however, are more complex than those of Hodes' System. In particular, the system allows programmers to embed user-interface customization rules in device descriptions. For example, it would allow a

programmer to embed a rule that a generator should keep the cassette navigation buttons of a stereo together and in a particular order (Figure 14). The generator would adhere to such rules when generating user-interfaces. Since these customization rules (or declarations) must be manually written, the PUC system supports semi-automatic generation.

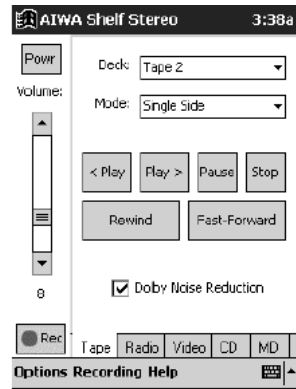


Figure 14. A portion of a stereo system user-interface generated by PUC [27]. Notice the cassette navigation buttons are specifically grouped together.

2.1.9 ICrafter

Like PUC and Hodes' System, Stanford's ICrafter[28] also supports the generation approach. However, it generates user-interfaces remotely from a client. In ICrafter, the generator runs on a machine that has a well-known location in a network of connected devices. To deploy a user-interface, clients access this generator and provide it with two important pieces of information: (1) a set of attributes that describe the client, which must at least include a list of UI languages it can support (e.g. HTML and Java Swing) and (2) a reference to the functional description of the target device. Given this information, the generator uses a declarative language to create a file that describes the user-interface for the client and target device pair. Clients download this file and then render the user-interface that it describes. In the HTML case, it is clear how this approach could work since HTML itself is a declarative language. The client simply uses a web-browser to render a web page that is the user-interface (Figure 15). However, the Java Swing toolkit is not inherently declarative. The builders of ICrafter thus built the Swing User-Interface Markup Language (SUIML), which is a declarative language for describing how Swing user-interfaces should look. Besides deploying single device user-

interfaces, ICrafter be used to also compose multiple devices. In the next section, we will describe how this is possible.

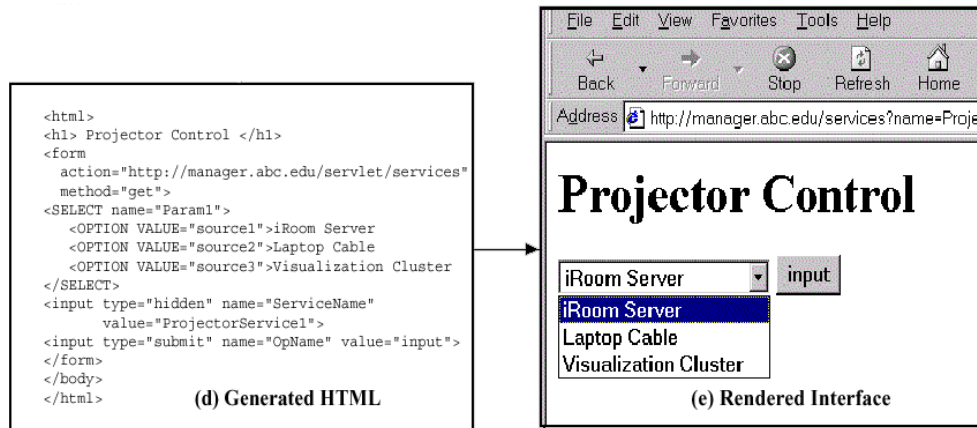


Figure 15. (left) the HTML generated for setting a projector input; (right) the rendered web page[28].

2.2 Deploying Multi-Device User-Interfaces

We extend the earlier general architecture to describe user-interface deployment in the multi-device case (Figure 16). In this case, the architecture includes composers, which use references from multiple devices to appropriately support some given set of composition semantics. These composers use user-interface deployers to deploy multi-device user-interfaces for the devices they compose. There are two current ways to organize composers within an infrastructure.

In one approach (Figure 16a), a composer uses the references of a set of devices to create a virtual device. This virtual device is represented in software as an integration of attributes and operations of multiple devices. In the ‘turn off all lights’ scenario, all the hallway lights in the building could be composed into a virtual device called an ‘all-hallway-lights-device’. This virtual device provides operations for simultaneously turning all the individual lights on and off. After the composer creates the virtual device, a user-interface deployer deploys a user-interface for it. A single-device user-interface deployer can be used in this case. It can be composition unaware since a virtual device, though representing multiple devices, simulates a single one.

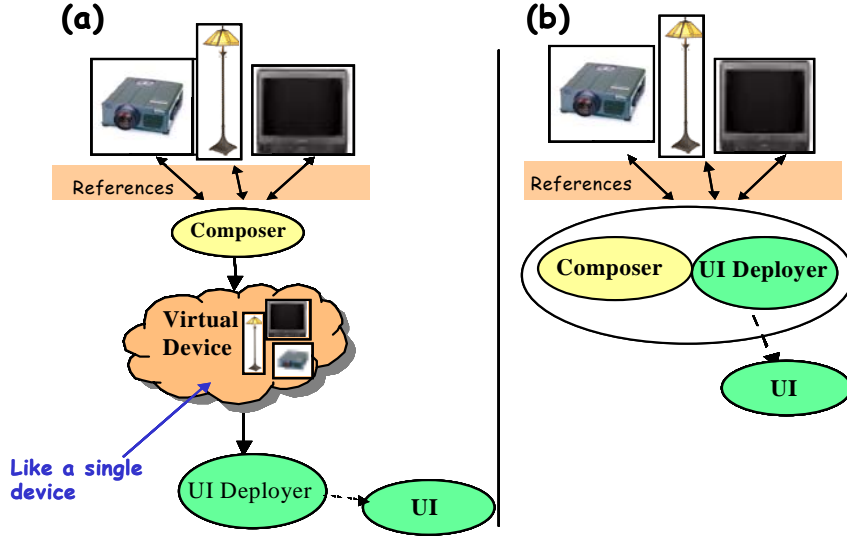


Figure 16. Two possible ways to organize composers.

In the other approach (Figure 16b), there is no notion of virtual devices. Instead, aggregation and user-interface deployment are tightly integrated. Meaning, a composer directly interacts with a user-interface deployer that is: (1) aware of the composer’s supported semantics and (2) capable of deploying user-interfaces for achieving those semantics. To support the lights scenario under this approach, a composer and user-interface deployer would cohesively work together to deploy the user-interface for turning off all the lights.

To show how these two high-level approaches can actually work, we will describe how specific infrastructures apply them. This discussion will fully illustrate the tradeoff between high-level support and composition flexibility mentioned in the previous chapter, thus motivating our *High-Level and Flexible Composition Hypothesis*.

2.2.1 Cougar and TinyDB

Cougar[5] and TinyDB[22] are two systems that were built to support queries for data over sensor networks. An example Cougar query is: *get the ‘current rainfall’ value of each sensor in Tompkin County*. Both Cougar and TinyDB implement mechanisms for performing such queries automatically and efficiently. They both work by requiring that devices advertise their attributes in distributed database relations and provide a relational

language to query these attributes. Device programmers, however, must write code that transfers state from device objects to database relations. This database-oriented framework of Cougar and Tiny follows the non-integrative composition approach described above. In essence, the two systems compose a group of distributed devices into single database—thus allowing a single user-interface program to be written for accepting arbitrary queries and returning results.

Both systems have limited flexibility in the composition semantics that they can support. They only compose devices using queries and do not provide frameworks for supporting other semantics described later.

2.2.2 Hodes' System

Hodes' System allows a user to interact with a set of devices through a single compound user-interface rather than their individual user-interfaces. For example, it can allow all the lamps in a conference room to share a single user-interface containing the commands for controlling them. Like Cougar and TinyDB, it also follows the non-integrative composition approach. To deploy compound user-interfaces, it generates user-interfaces from manually generated XML-based descriptions of compound (virtual) devices that encapsulate descriptions of multiple devices. The 'conference room lights' virtual device could be described as the following:

```
<service name = 'Conference Room Lights'>
  <label>Conference Room</label>
  <addrspec>sn011.unc.edu/0001</addrspec>
    <service name = 'lamp1'>
      <label>Lamp 1</label>
      <addrspec>sn011.unc.edu/0001</addrspec>
      <method name = 'on'></method>
      <method name = 'off'></method>
      <method name = 'dim'></method>
      <method name = 'brighten'></method>
    </service>
    <service name = 'lamp2'>
      <label>Lamp 2</label>
      <addrspec>sn011.unc.edu/0002</addrspec>
      <method name = 'on'></method>
      <method name = 'off'></method>
      <method name = 'dim'></method>
      <method name = 'brighten'></method>
    </service>      . . .
  </service>
```

A generated user-interface for the ‘conference room lights’ device would resemble the one shown in Figure 17, which vertically places the individual user-interface of each lamp on top of one another.

Hodes’ System supports a limited set of composition semantics. It does not support device queries as Cougar and TinyDB do. Further, it does not support other semantics demonstrated by the other systems below.



Figure 17. A compound UI for a set of lights.

2.2.3 Palm/Pocket-PC IR Programs

Beyond the ability to create single device user-interfaces, these programs allow a person to build compound user-interfaces as supported by Hodes’ system. They provide wizards for users to merge the single device user-interfaces they design (as described in 2.1.1) to form compound user-interfaces. In addition, they typically allow users to create macro buttons that automatically invoke specific sequences of commands from multiple devices. For example, they could allow a person to create a ‘watch DVD button’. When pushed, the button invokes six different operations that prepare a TV, DVD, and receiver for watching a movie:

- 1) Turn on the TV
- 2) Set TV to DVD video input channel
- 3) Turn on the receiver
- 4) Set the receiver to DVD audio input
- 5) Turn on the DVD player
- 6) Open the DVD player’s disc tray

2.2.4 WebSplitter

WebSplitter[13] can compose devices together to present different types of content contained in a set of web pages. For instance, it can compose an audio system, projector, and other display devices to present a multimedia web presentation consisting of visual frames (images and text) and audio content. The display devices show the content slides, navigation buttons, and notes of the presentation while the audio player plays the audio (Figure 18).

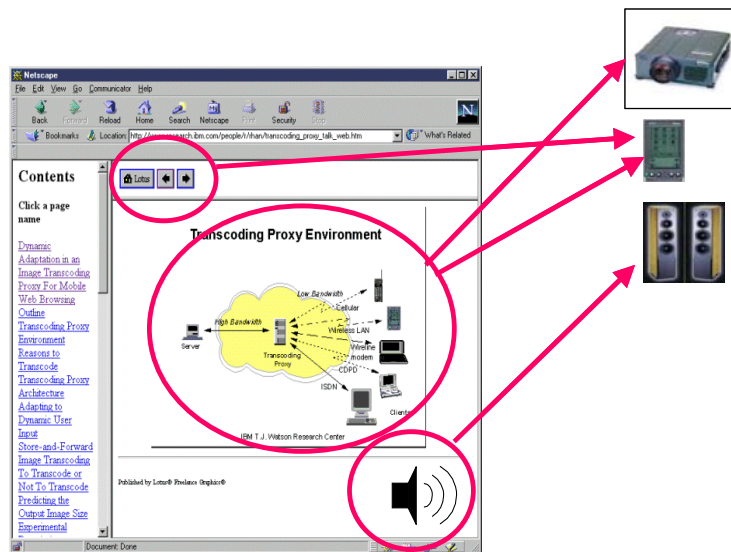


Figure 18. A Websplitter presentation in which audio is sent to a stereo and frames are shown in display devices [13].

As a speaker navigates through this web presentation, WebSplitter automatically delivers the URLs of content in each page to the appropriate devices.

In order to properly map or 'split' the content of a web page to their associated devices, it requires users to write XML-based policy files. These files specify mappings between the content of each page and the kinds of devices that should receive them. The policy file for our example presentation could contain syntax such as the following:

```
<cmdb:device name = "projector">
  <cmdb:taglist>
    presentation, head, title, nav_bar, slides, picture
  </cmdb:taglist>
</cmdb:device>
```

```

<cmdb:device name = "sound system">
  <cmdb:taglist>
    audio
  </cmdb:taglist>
</cmdb:device>

<cmdb:device name = "cellphone">
  <cmdb:taglist>
    nav_bar,
  </cmdb:taglist>
</cmdb:device>

```

It specifies that: (1) the projector should receive all content except the speaker's private presentation notes, (2) the audio system should receive all presentation audio, and (3) the speaker's cell phone should receive the navigation bar for controlling the presentation's pace. For each page in the presentation that the speaker visits, WebSplitter refers to the defined mappings in the policy file to correctly direct the page's content. WebSplitter's set of supported composition semantics is limited. It cannot achieve any of the semantics facilitated by the other systems described above and some below.

2.2.5 ICrafter

ICrafter provides a general framework for actually writing multiple composers supporting different composition semantics. It is unlike the systems described above which offer preprogrammed composers that support a fixed and limited set of semantics. To provide a general framework for composition, ICrafter's composers work in terms of the programming interfaces of devices rather than their classes. Since programming interfaces are more general than classes, this approach provides a way for single composers to compose families of heterogeneous devices.

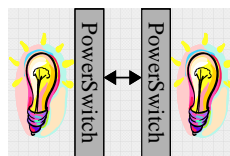


Figure 19. A lights composition.

To illustrate, consider a system with lights shown in Figure 19. Suppose that the two lights implement a `PowerSwitch` interface that declares a `power()` method for turning the lights on and off. A programmer can write a `PowerSwitchAll` composer for this

programming interface that provides the ability to simultaneously turn these lights on and off. This composer provides the system with a regular expression describing that it composes devices implementing the `PowerSwitch` interface (Table 1). In turn, the system matches the composer with the two lamps and presents the match to a user. A user’s selection of this match results in the `PowerSwitchAll` composer generating a user-interface consisting of a ‘power all lights’ button. When the button is pushed, the composer invokes the well-known `power()` method of each lamp. This example shows that ICrafter follows the integrative composition approach mentioned earlier. Composers offer their own generators for dynamically creating user-interfaces that are specific to their supported semantics and users’ target devices.

Regular Expression	Composer
{PowerSwitch*}	PowerSwitchAll

Table 1. An example composer registry.

Figure 20 depicts a different composition scenario, which involves transferring images in a camera to a display device for viewing. Let us assume that the camera and display device implement a `DataProducer` and `DataConsumer` interface respectively. The `DataProducer` interface declares a `produce()` operation, which returns a value to transfer, and the `DataConsumer` interface declares a `consume()` operation, which accepts the value. A programmer can now write a `DataPipe` composer that allows the camera and display device to exchange data. This composer provides the system with a regular expression describing that it composes devices implementing the `DataProducer` and `DataConsumer` interfaces (Table 1).

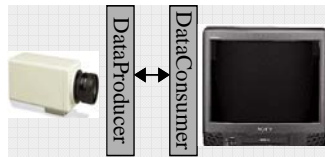


Figure 20. An example composer registry.

Regular Expression	Composer
{DataProducer, DataConsumer}	DataPipe

Table 2. An example composer registry.

The system would match the camera and display device programming interfaces with the `DataPipe` composer and present the match to a user. A user's selection of this match results in the `DataPipe` composer generating a user-interface for invoking the transfer operation. Once the user invokes the operation on the user-interface, the composer calls the well-known methods of its associated programming interfaces to achieve the image transfer. That is, it makes a call that passes the value returned from `camera.produce()` as an argument to `display.consume()`.

An issue with performing data transfers is how `DataProducer` and `DataConsumer` interfaces declare the data they exchange. Two options are to declare data as: (a) a generic object or (b) a programmer-defined type. In Java, the class `Object` demonstrates this notion of a generic data type. All classes in Java are subclasses of `Object` and can therefore be typecasted to it. Using generic objects, the two programming interfaces would be:

```
public interface DataProducer {
    public Object produce();
}
public interface DataConsumer {
    public void consume(Object x);
}
```

Here, the producer returns a value of type `Object` and the consumer accepts a value of that same type. An example of using programmer-defined types is below, in which the consumer and producer specifically exchange a `Picture` object:

```
public interface PictureProducer {
    public Picture produce();
}
public interface PictureConsumer {
    public void consume(Picture x);
}
```

These two options raise a subtle tradeoff a programmer must make between type flexibility and programming cost.

The benefit of the generic approach is that it would require implementing only one composer, a truly generic `DataPipe` composer, to accomplish data transfers. One drawback is that all consumers are able to arbitrarily match with all producers because

they all produce and consume the same generic type. When interacting with many devices, this approach could result in lists of many false-positives—that is, matches between devices that cannot exchange data. An example of a false positive is a match between a camera that only produces picture objects and an alarm clock that consumes time objects. Another drawback of using generic objects is that a device can only consume or produce one kind of data because forcing the generic type does not allow overloading of the `consume()` and `produce()` methods in the programming interface declarations. Therefore, the camera could not independently produce URLs to both pictures and recorded video.

Supporting programmer-defined types in programming interfaces reduces the production of false positives because it allows consumers and producers to be matched by the types they exchange. It also allows overloading of the `consume()` and `produce()` methods so that devices can exchange more than one data type. However, it incurs the costs of writing many composers that are specific to the data types that devices can exchange. To illustrate, it requires writing separate `PicturePipe` and `VideoPipe` composers so that the camera can transfer two different kinds of data types.

The `PowerSwitchAll` composer raises another tradeoff the programmer must make in writing composable programming interfaces. Which programming interfaces should a device implement? Two extreme options are a programming interface for all operations of a device or a programming interface for each operation. The first piece of code below is an example of the former approach while the second demonstrates the latter:

```
1)    public interface Light {
        public void power();
        public void dim();
        public void brighten();
    }

2)    public interface PowerSwitch {
        public void power();
    }
    public interface DimSwitch {
        public void dim();
    }
    public interface BrightenSwitch {
        public void brighten();
    }
```


In the first case, it is not possible to write a generic composer for devices implementing different programming interfaces even if they have common operations. For example, it is not possible to write a composer for simultaneously invoking the power operations of a TV and a light, since they provide different sets of operations. The latter approach overcomes the limitations of the earlier. However, it leads to a proliferation of programming interfaces and associated composers. Supporting the ‘Power All’, ‘Dim All’, and ‘Brighten All’ operations on the lights requires three separate programming interfaces and composers. An intermediate approach that defines programming interfaces for subsets of operations offers intermediate degrees of composition flexibility and programming cost of these two extreme approaches.

2.2.6 Speakeasy

Speakeasy (also called Obje) [9] is another system that provides a generic composition framework. It also uses a programming interface based approach. However, it avoids two of the problems of ICrafter—proliferation of programming interfaces and false positives.

To avoid this interface proliferation problem, Speakeasy adopts the notion of generic programming interfaces. To illustrate, devices that consume or produce data would implement a generic ‘data transfer’ programming interface. This programming interface does not contain information about the specific data type the devices can exchange and whether they consume or produce the value of that type. As our discussion of ICrafter shows, generic programming interfaces can result in false positives when composing devices of matching programming interfaces. However, this problem is associated with systems that support automatic matching of producers and consumers. Speakeasy, on the other hand, takes a manual matching approach. It provides a user-interface in which users, themselves, select and appropriately connect the devices of matching programming interfaces. Thus, it relies on users to not make false positives. To assist users, devices must implement operations that return objects that indicate the values (including type descriptions) they can exchange. For example, a digital camera would implement an operation that returns an object indicating that it stores images as JPEGs. Also, a display

device would implement an operation returning an object indicating that the device only displays GIF images. A user would discover that these two devices are incompatible for data transfer by comparing their supported picture formats.

The builders of Speakeasy performed user-studies to measure the burden of the manual connection approach on users. These studies show that for typical device users, this approach can be too low-level and difficult. To address this problem, the builders intend to offer a mechanism that allows technically savvy users within a site (e.g. office building) to store and publicly distribute templates of the compositions they make to others. Non-savvy users could simply load these templates onto their clients and avoid connecting devices themselves. For example, a non-savvy presenter could retrieve a template for giving presentations in a particular conference room. This template automatically composes the lights, audio equipment, and projector in ways that prior presenters have found useful when giving a presentation. With the template, the presenter could simply provide the name of the file that contains of the presentation slides. All other configuration processes are automated.

Large sites, such as office buildings and college campuses, will likely have ‘gurus’ (e.g. system administrators and facilities managers) that are capable of making such templates that compose their publicly accessible devices. However, this assumption cannot be made for a small site, such as a family home, which is more likely to have non-technically savvy users. Further, since the arrangement and use of devices can vary from home to home, it is not clear if and how much households can share templates.

Chapter 3: Analysis of Various Approaches

As discussed in the previous chapter, there are various approaches to deploying software-based user-interfaces for devices. At the highest level are the predefined and generation approaches. The predefined approach involves using a factory to supply pre-existing user-interface code. This factory can be located at a user's client machine, the target device, or some third-party machine. The generation approach is the converse of the predefined approach because it does not require devices to be loaded with pre-defined user-interface code. Instead, a user-interface generator, residing on the client or a remote machine, dynamically creates an appropriate user-interface by using information extracted from a device's functional description. The generator can support a semi-automatic and/or fully automatic approach to creating user-interfaces. Under the semi-automatic approach, the generator accepts manually written declarations consisting of rules to follow when creating a user-interface. In the fully automatic case, the generator directly creates a user-interface without the use of such declarations.

In this chapter, we fill an existing void in this area by systematically evaluating these existing approaches. Our work offers several contributions: (1) an identification of several metrics for comparing the approaches, (2) a qualitative analysis of the approaches based on the identified metrics, (3) a quantitative analysis that verifies our qualitative arguments and quantifies the differences between the approaches. These contributions subsequently lead to a proof of the *Uniqueness Hypothesis*: each existing approach offers a set of unique benefits, thus providing a reason for why it exists.

3.1 Overview of Metrics and Setup

Below are five useful evaluation metrics we have identified:

- 1) *User-Interface Flexibility* – range of user-interfaces an approach can support
- 2) *Programming Costs* – amount of code required to deploy a user-interface

- 3) *Maintenance Costs* - programming time and resources required to support and update user-interface code
- 4) *Efficiency* – time and storage space costs of an approach
- 5) *Device Binding Time* – time a client must learn about (or bind to) a device in order to deploy a user-interface for it.
- 6) *Deployment Reliability* – the level of guarantee an approach offers in deploying a user-interface

As just mentioned above, these metrics allow us to make a mix of qualitative and quantitative comparisons of the various approaches. 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 author owns all of the devices except the projector, which is found in one of the conference rooms in our department's building.

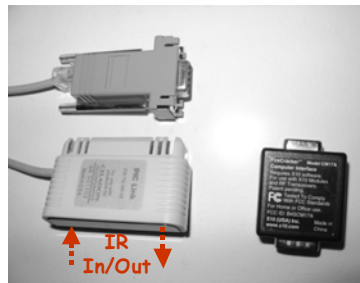


Figure 21. (left) a *Celadon PIC Link* IR module (right) an *X10 FireCracker CM17A* module

For each device, we created a Java RMI (proxy) object representing its functionality on a desktop PC {Windows XP, 1.68GHz Pentium, 512MB, wired LAN (100Mbps) connection}. 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 (Figure 22). 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 (Figure 21). 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. As a result, the methods do not return any values or require any parameters. To illustrate this command-to-method relationship, consider a `power()` method invocation on the receiver's proxy object. The method triggers the IR module to emit the signal previously recorded from us pushing the power button on the actual receiver remote. The receiver would treat this signal as if it came from a compatible traditional remote.

```
public interface Receiver{
//commands
    public void power();
    public void sleep();
    public void video1();
    public void video2();
    public void video3();
    public void dvdORld();
    public void tvORsat();
    public void aux();
    public void mdORTape();
    public void cd();
    public void tuner();
    public void phono();
    public void _51CH_();
    public void AFD();
    public void _2CH_();
    public void mode();
    public void analogDirect();
    public void cinemaStudio();
    public void bassBoost();
    public void mute();
    public void volumeUp();
    public void volumeDown();
    public void ChORPresetUp();
    public void ChORPresetDown();
    public void EqORTone();
    public void testTone();
    public void rearUp();
    public void rearDown();
    public void centerUp();
    public void centerDown();
    public void _0_0;
    public void _1_0;
    public void _2_0;
    public void _3_0;
    public void _4_0;
    public void _5_0;
    public void _6_0;
    public void _7_0;
    public void _8_0;
    public void _9_0;
    public void shift10();
    public void enter();

//state
    public boolean getPowered();
    public void setPowered(boolean _powered);
    public int getBass();
    public void setBass(int _bass);
    public int getTreble();
    public void setTreble(int _treble);
    public int getRfBalance();
    public void setRfBalance(int _rfBalance);
    public int getLfBalance();
    public void setLfBalance(int _lfBalance);
    public int getCBalance();
    public void setCBalance(int _cBalance);
    public int getRrBalance();
    public void setRrBalance(int _rrBalance);
    public int getLrBalance();
    public void setLrBalance(int _lrBalance);
    public int getSubBalance();
    public void setSubBalance(int _subBalance);
    public int getRfVolume();
    public void setRfVolume(int _rfVolume);
    public int getLfVolume();
    public void setLfVolume(int _lfVolume);
    public int getCVolume();
    public void setCVolume(int _cVolume);
    public int getRrVolume();
    public void setRrVolume(int _rrVolume);
    public int getLrVolume();
    public void setLrVolume(int _lrVolume);
    public int getSubVolume();
    public void setSubVolume(int _subVolume);
    public int getVolume();
    public void setVolume(int _volume);
    public double getChannel();
    public void setChannel(double _channel);
    public String getInput();
    public void setInput(String _input);
    public boolean getTone();
    public void setTone(boolean _tone);
    public boolean getBassBoost();
    public void setBassBoost(boolean _bassBoost);
    public String getMode();
    public void setMode(String _mode);
    public boolean getMuted();
    public void setMuted(boolean _muted);
};

public interface Lamp {
//commands
    public void on();
    public void off();
    public void dim();
    public void brighten();

//state
    public int getBrightness();
    public boolean getPowered();
    public void setPowered(boolean _powered);
}
```

Figure 22. Sample programming interfaces (for the receiver and lamp).

The lamp is the only device that we networked using the X10 protocol. This protocol has fixed and generic signals for powering and dimming arbitrary lamps. Thus, we did not need to perform any signal recording.

Beyond representing our devices' commands in the proxy objects, we also incorporate their state. Including state allows the proxy objects to more closely emulate networked versions of the actual devices. Moreover, it allows us to consider the differences between deploying command-only and command-and-state-based user-interfaces in our evaluation. To represent device state in the proxy objects, we searched the on-board panels and menus of each device for status information to represent as Java Bean properties. This representation of state allowed us to use ObjectEditor to generate GUIs displaying the state of each device. Figure 22 illustrates our use of this state representation by showing the programming interfaces for the projector and lamp proxy objects. Since today's devices are generally unable to send output messages, even using IR, they are incapable of notifying external agents of their state changes. As a result, we included code in the proxy objects that keeps the property values consistent with the actual state of their associated devices. If a user turns on the lamp, for example, the `powered` property would be set to `true` before the `Lamp.on()` method terminates. Invoking `getPowered()` immediately after `Lamp.on()` would therefore return `true`. Vice versa, if the user turns the lamp off, then `getPowererd()` would return `false`.

3.2 User-Interface Flexibility

Under the predefined approach, programmers are able to handcraft arbitrary kinds of user-interfaces. A user-interface generator, on the other hand, is limited to creating the kinds of user-interfaces it was programmed to deploy. Thus, in theory, this approach has lower user-interface flexibility than the predefined approach. In practice, however, the flexibility of the generation approach depends on the domain being considered.

We first consider a domain of user-interfaces based on the design of traditional remote controls, which have been a principal means for interacting with devices for many years. A class of user-interfaces that logically belong in this domain consists of GUIs that are built to directly mimic traditional remote controls. Using ObjectEditor, we evaluated the

ability to fully automatically generate such user-interfaces for the six devices we networked.

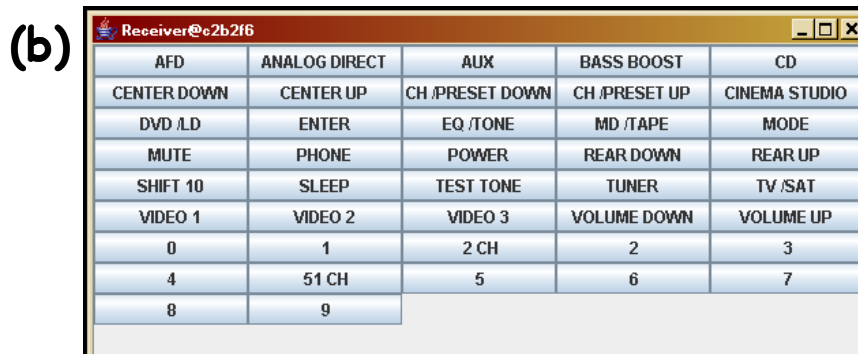
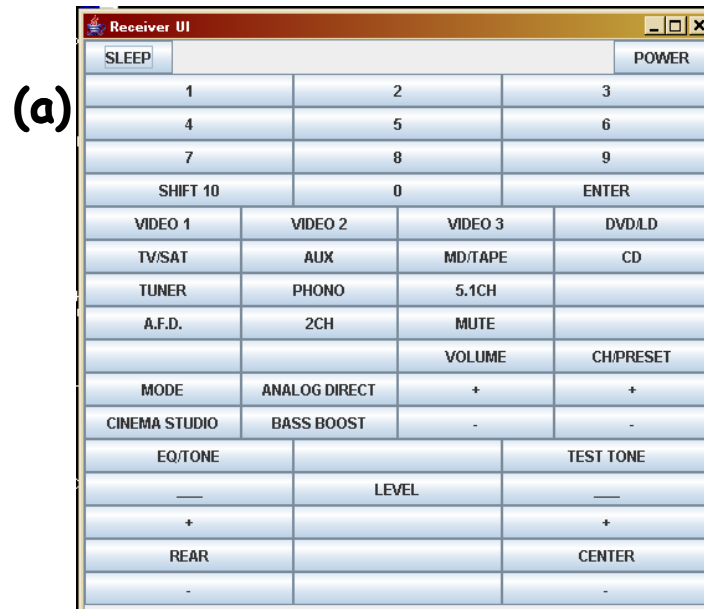


Figure 23. Command-only receiver GUIs written using Java Swing: (a) a predefined GUI that mimics the device’s remote control and (b) a GUI generated fully automatically by ObjectEditor.

Figure 23 shows the handcrafted remote-control-mimicking GUI that we built for the receiver and the GUI fully automatically generated by ObjectEditor. There are major differences between the two GUIs. The handcrafted GUI follows conventions actual found in the receiver’s remote control. For example, it arranges its buttons in a ‘number pad’. It also groups commands that perform related functions together. For instance, it places the mute button near the volume buttons. ObjectEditor, which has no inherent notion of how to generate a GUI that mimics receiver’s remote control, simply places

buttons in alphabetical and then numerical order. This type arrangement is not consistent with the remote control. For some cases, though, this ordering places related buttons next to one another if they share prefixes (e.g. Volume Down with Volume Up, Rear Down with Rear Up). It also places the number buttons together, though not in a ‘number pad’ arrangement. Appendix A contains snapshots of the handcrafted and fully automatically generated GUIs for the other devices that we networked. Basically, all fully automatically generated GUIs exhibit limitations that are similar to those of the receiver case.

It is possible to overcome some of this flexibility limitation of fully automatic generation by supporting a semi-automatic approach. As Figure 23 illustrates, ObjectEditor and the handcrafted remote-control-mimicking GUIs follow a grid-based layout for arranging buttons. ObjectEditor can accept manually written declarations that describe the absolute position and label of each button on a grid. Such declarations thus allow the generator to create user-interfaces whose buttons follow the same ordering as the handcrafted ones.

A problem, though, arises from trying to support button labels that are identical to the handcrafted GUIs. In the handcrafted receiver GUI (Figure 23a), notice that the ‘CH/PRESET’ up and down button are respectively labeled using a ‘+’ and ‘-’. However, in the generated user-interface, they are labeled as ‘CH/PRESET UP’ and ‘CH/PRESET DOWN’. These could respectively be replaced with a ‘+’ and ‘-’ by giving appropriate user-defined declarations. This facility raises a new problem in that with such replacements, the CH/PRESET buttons would no longer indicate their functionality to users. In other words, there is no information that users would see to know what the ‘+’ and ‘-’ buttons perform. In the handcrafted GUI, their purpose is clear because there is a ‘CH/PRESET’ label above the two buttons that indicates their purpose. This label differentiates the CH/PRESET up and down buttons from those for VOLUME. It is not possible to arbitrarily insert such labels, or any new user-interface component, in an ObjectEditor-generated user-interface using declarations. This ability is low-level and requires manually changing the generator’s code. Thus, it demonstrates the flexibility

limitation of the semi-automatic (or declaration-based) generation in the domain of remote-control-mimicking GUIs.

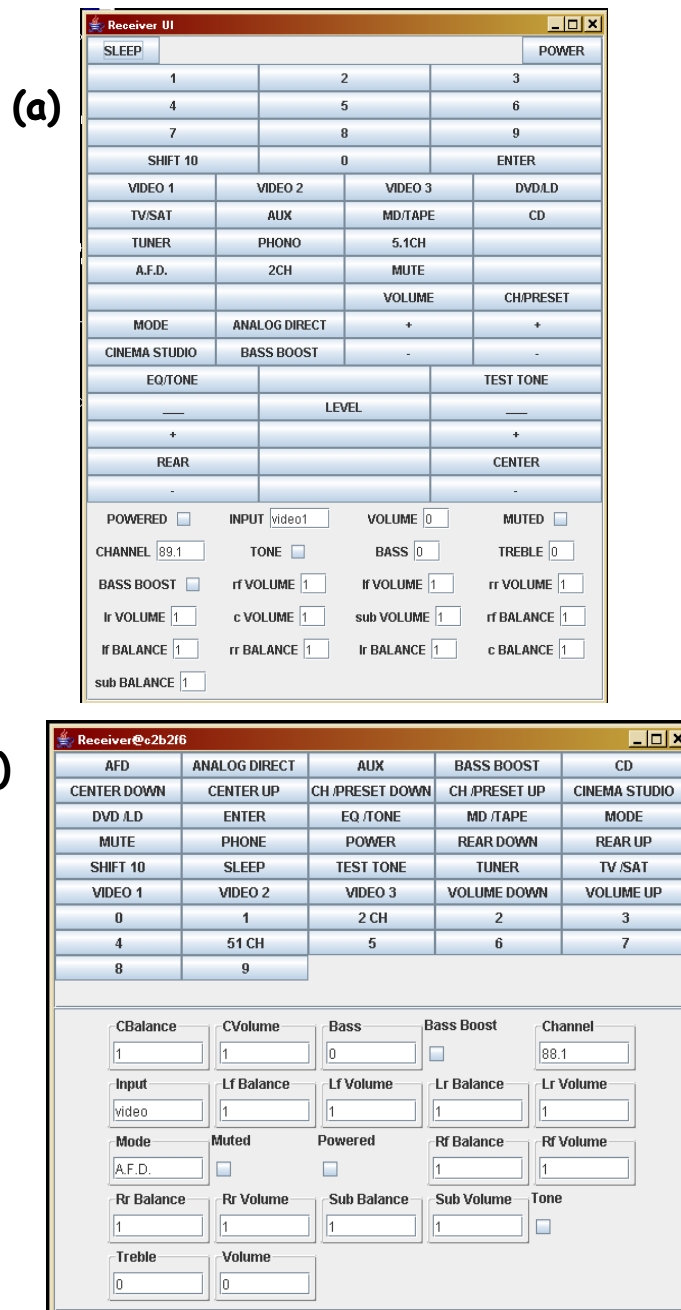


Figure 24. Command and state-based receiver GUIs: (a) predefined GUI and (b) generated by ObjectEditor fully automatically.

Let us now consider remote-control-mimicking GUIs that also incorporate device state. In particular, we consider GUIs that display primitive typed state since all of the state property types of all of our devices are primitive. 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 simply extending the standard grid-based layout of buttons to state display. Figure 24a illustrates this design by displaying the receiver's handcrafted remote-control-mimicking 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 receiver's 'powered' property value is displayed by a checkbox. An unchecked box means that the receiver is off, otherwise, it is on.

As shown in Figure 24a, the state widgets of the GUIs are all placed in their bottom panel using a grid layout. The entire state panel of the receiver's GUI, for example, is based on a 4 by 5 grid. Next to this handcrafted GUI (Figure 24b) is the corresponding GUI that ObjectEditor generated fully automatically. Recall from our description of ObjectEditor, that the generator can also create GUIs displaying device state. For each property in a device object, the generator maps it to a widget for displaying its value. Figure 24 shows that the generator can automatically create state representations that are similar to those of the handcrafted GUIs. Further, the generator also places the widgets on a panel at the bottom of the GUI using a grid-based layout. However, there are some clear differences between the two state panels. First, as before, the two GUIs order widgets differently. For example, the 'muted' and 'volume' property widgets are next to one another in the handcrafted GUI and not in the generated one. Second, in the handcrafted GUI, the labels displaying the names of each property are placed in to the left of the value display. In the generated GUI, they are on top of the values. Third, the sizes of the textboxes displaying numerical and string based properties differ between the two GUIs. Notice that the textboxes of the generated GUIs are wider. Finally, five, the dimensions of the state grid differ. ObjectEditor places the receiver GUI's state widgets in 5 by 5 grid. In Appendix A, we present snapshots of the handcrafted and fully automatically generated GUIs (with state) for the other devices that we networked.

Basically, all fully automatically generated GUI exhibit similar limitations as just described for the receiver case.

To present the state widgets in the same manner as the handcrafted GUI, `ObjectEditor` can accept manually provided declarations describing their absolute position on a grid. Moreover, the generator can accept: (a) a single declaration stating how to position the labels of all widgets or (b) a declaration for each widget that describes its individual label position. Furthermore, the generator can also accept: (a) a single declaration stating how size all of the textboxes or (b) a declaration for each textbox that describes its individual size. As a result, in displaying primitive typed state, the generation and predefined approaches share the same flexibility for the devices we considered. It is unclear whether semi-automatic generation can equally support other kinds of state-based representations. In particular, we are uncertain about its ability to support structured-type state, which we do not consider here. The concern is based on the fact that structured-types can require complex representations in a user-interface.

Now, let us consider the ability to generate remote-control-based SUIs. For our six devices, we handcrafted the SUIs using the Java Speech API and the IBM ViaVoice 9 speech recognition/synthesis engine. The SUIs were built to allow a user to invoke methods on a device's proxy object by simply speaking their names. Recall that these methods correspond to commands displayed as buttons on the traditional remote control. Hence, there is a direct mapping between the SUIs and their corresponding traditional remote controls.

Once loaded, each SUI notifies the user to 'start talking'. The user can then either say: (1) "get commands" to hear the names of possible commands to invoke on the target device or (2) directly say the name of the command to invoke (e.g. "mute"). Basically, the grammar (or set of acceptable words) of the dialogue consists of the names of each device command and the phrase "get commands". Our devices' proxy objects all have parameterless methods. As a result, we did not have to support the ability to enter parameter values in the dialogues.

To test the ability to fully automatically generate the handcrafted GUIs described above, we built an experimental SUI generator for Java objects. Like the handcrafted SUIs, we coded the SUI generator using the Java Speech API and IBM ViaVoice 9. We were able to build the generator to follow the same dialogue structure as the handcrafted SUIs. To create this structure, it extracts a device object’s method names using Java Reflection and then adds them to the input grammar (set of acceptable words) of the recognizer. To invoke

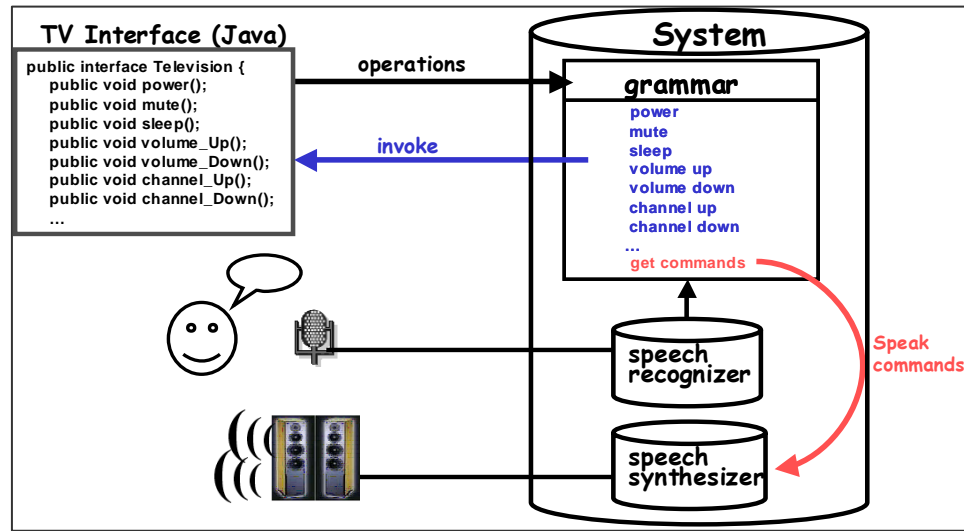


Figure 25. A depiction of our experimental SUI generator.

a method on a device, users simply say its corresponding name. By default, the generator also adds the phrase ‘get commands’ to the grammar. Users can say this phrase to hear a list of commands extracted from the target device. Similar to the handcrafted SUIs, users cannot begin speaking any phrases until generator notifies them to ‘start talking’—i.e. the generation algorithm is done. Figure 25 depicts this algorithm illustratively, and below, we describe it using pseudocode:

```

t = a reference to the target device object
G = an initially empty grammar
RECOGNIZER = the recognizer
SYNTHESIZER = the synthesizer

generateSUI(t, G, RECOGNIZER, SYNTHESIZER) {
    M = getMethodNames(t)
    for each method name (x) in M
        G.insert(x)

```

```

        G.insert("get commands");
        RECOGNIZER.start();
        SYNTHESIZER.speakText("start talking");
    }

```

Once a user speaks a phrase (p), the system executes the following:

```

processPhrase( $p, G, t$ ) {
    if  $G.contains(p)$  {
        if  $p.equals("get commands")$  {
             $M$  = a set consisting of  $t$ 's method names
            for each method name ( $x$ ) in  $M$ 
                SYNTHESIZER.speakText( $x$ );
        }
        else {
             $method$  = actual method with name  $p$ 
             $t.invoke(method)$ 
        }
    }
}

```

The above shows that it is possible to generate the remote-control-based SUIs fully automatically. That is, our generator does not require the support of manually written declarations to create the SUIs. As we did with GUIs, we will not continue on to evaluate the ability to generate remote-control-based SUIs that present state. The reason is that, unlike for the GUIs, there is no clear way for deriving a way to present state in SUIs.

There are unlimited kinds of other SUIs and GUIs that we could further consider in our flexibility evaluation. As discussed in the Introduction, such user-interfaces could be ‘far beyond remote-control-based’. They could include advanced features such as record/replay, concurrency, failures, and disconnected interaction/synchronization. It is not clear how well such user-interfaces can be generated, even with the aid of manually provided declarations. Thus, it is in this space where the tradeoff between the automation and user-interface flexibility of generation mainly exists. The next section expands on this tradeoff by evaluating the programming costs involved in the different approaches.

3.3 Programming Costs

User-interface programming can be a cumbersome task. Surveys show that implementing a user-interface of a conventional application requires more code than implementing its functionality [25, 37]. The user-interface of a network device may be

even more difficult to implement, for two reasons. First, it is remote to the device, and it must address network issues. Despite the desire for transparent remote access, no practical system offers it. In fact, it is believed by some that no practical system can, because of the need to address latency, partial failure, concurrency, and a different memory model [40]. Second, as suggested earlier, the user-interface might offer several new and potentially useful kinds of features that are missing in traditional remote controls such as record/replay and disconnected interaction/synchronization. These features are very difficult to implement, as evidenced by the fact that several commercial applications such as spreadsheets and drawing tools do not offer them, even though their usefulness goes beyond our domain (in particular, they are useful in collaborative or mobile access to data [6, 24]).

In the user-interface generation approach, programming costs are paid in writing generator code for each kind of user-interface toolkit (e.g. Java Swing, Java Speech API) supporting one or more devices. Once a generator is built, there is no cost in creating user-interfaces fully automatically. With the predefined approach, device programmers must manually implement user-interfaces for each kind of device and toolkit (Figure 26).

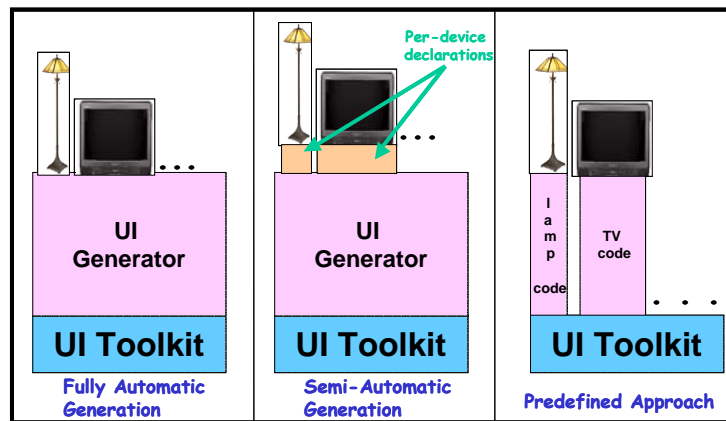


Figure 26. UI generation vs. the predefined approach.

We counted the number of lines of code required to manually implement the remote-control-based user-interfaces mentioned in the previous section (Table 3). Table 3 shows the results and compares them with the number of lines of declarations needed for semi-automatic generation. Recall that ObjectEditor cannot generate the exact remote-control-mimicking GUI semi-automatically. Thus, for the programming costs of semi-automatic

GUI generation, the values represent the number of lines required to produce GUIs that are as close to the handcrafted ones as possible. To appropriately position a button and state-widget, ObjectEditor simply requires a one-line declaration. For each state widget, the generator also requires a line for specifying the placement of the accompanying property label. It additionally needs a line that for each string and numeric based property of a device to indicate the size of the corresponding textbox. The reason is that the textboxes in a device's handcrafted GUI may not all the same size. The 'input' property's textbox in the receiver's user-interface, for example, is larger than that of the volume property (Figure 24a). If the textboxes all shared the same size, the generator would only need a single declaration indicating the size for all textboxes.

Device	Number of lines of UI code					
	GUIs				SUIs	
	Remote-Control-like GUI		Remote-Control-like GUI with State		Semi-automatic generation	Predefined
	Semi-automatic generation	Predefined	Semi-automatic generation	Predefined		
Receiver	42	287	66	428	0	176
DVD Player	38	316	54	414	0	168
Lamp	4	94	8	110	0	102
Projector	23	254	31	283	0	138
TV	25	321	36	385	0	142
VCR	40	247	47	288	0	172

Table 3. Number of lines of user-interface code used for each device

Unlike ObjectEditor, the SUI generator did not require any lines of manually provided declarations in order to generate its target remote-control-based user-interfaces. This zero line-cost of fully automatic generation is a significant benefit because of the following point. Although manually implementing a user-interface can have a relatively small (non-zero) line-cost, a programmer must actually spend the time doing it. Consequently, the predefined approach does not always guarantee a user-interface for any given device. The zero-line cost of fully automatic generation inherently guarantees a user-interface.

3.4 Maintenance Costs

A deployment approach should be able to respond to a rapidly changing set of devices, mobile clients, and user-interface paradigms. Therefore, programmers must continuously write new code and organize components of a deployment infrastructure in order to support change. In general, an approach that offers low programming costs also makes it easy to support change.

3.4.1 Predefined vs. Generation

Research shows that user-interface code tends to be the least portable part of all application code [10]. Consequently, the code for generators and predefined user-interfaces may not be compatible with future toolkits. Given a new toolkit, the predefined approach could require coding new user-interfaces for each device. On the other hand, the generation approach could require coding a new generator, which creates user-interfaces for an arbitrary number of devices. Because the number of devices needing new user-interfaces can be high, qualitatively, the maintenance cost of the predefined approach is greater than that of the generation approach.

User-interface generation has the added advantage of being able to easily support forms of interactions that were not initially intended for a device. To illustrate, we could incrementally add foreign language support to ObjectEditor generator. The generator could access a translator program to convert a method name in one language to its corresponding name in another language. The predefined approach also allows additional user-interface paradigms, but it requires changing the code of many existing user-interfaces or implementing new ones.

3.4.2 Client-Factory and Third-Party Factories vs. Other Approaches

In the predefined approaches, the removal of a device should also result in the removal of its user-interface code. It is difficult to perform this task in the client-factory and third-party-factory approaches because this code is separated from devices. Therefore, the approaches require additional maintenance to track the locations of user-interface code in order to completely remove it. In the device-factory approach, the user-interface code is inherently removed with the device. Hence, the maintenance cost is lower. The fully

automatic generation approach does not face this problem because there is no per-device user-interface code. Semi-automatic generation, however, has per-device user-interface code. It is no better, in this respect, than the client and third-party-factory approaches.

3.5 Efficiency

Below are four efficiency factors to be considered in deploying user-interfaces:

- Space Costs:
 - *Device Space*: Storage used at the device.
 - *Client Space*: Storage used at the client.
- Time costs:
 - *Deployment Time*: Time required to deploy a user-interface.
 - *Operation Invocation Time*: Time required to invoke an operation on a device. That is, the time from when a user requests the operation to the time when the device's object starts the operation.

The space used at the third-party machine is not an issue since we believe that the machine would be at least as powerful as modern desktops and would be connected to disks. However, the space used at the device and client is important because most devices and mobile computers are typically required to be small and/or inexpensive. Cell phones and networked lights, for example, are bound to have little storage. 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” [33]. Similarly, the time costs are important because: (a) devices and mobile clients have low processing power, (b) the alternative, traditional remote controls, have no deployment cost, and transmit user's intentions to the device at infrared speed, and (c) users get frustrated with high system response times [34].

3.5.1 Space Costs

The device-factory approach consumes the most device space because, unlike the other approaches, user-interface code is actually stored on devices. On the other hand, the generation, client-factory, and third-party-factory approaches consume the least amount of device space since they require no user-interface code on the machine hosting device objects. We ignore the insignificant space taken by customization code required by semi-automatic generation.

The remote-factory and remote generation approaches consume the least client space because they do not require clients to store any user-interface code. In comparing the client-factory and the client-side generation approaches, the client space cost depends on the number of devices with which a user will interact. This number can be high enough such that the space consumed by the required handcrafted user-interfaces is greater than the space consumed by a single generator. Conversely, it can be low enough such that the generator consumes more space than a small set of handcrafted user-interfaces.

To provide a quantitative dimension to this evaluation, we measured the space consumed by some user-interface generators and the code of the device user-interfaces we handcrafted. `ObjectEditor` consumes over 898,000 bytes, which is much higher than the total space consumed by the code of all the handcrafted GUIs (Table 4). However, recall the generator’s complexity. It can create user-interfaces for applications beyond the domain that we have considered for devices. This additional functionality consumes a significant part of the total space of the application. Thus, a direct comparison of `ObjectEditor`’s space and with the space needed to store predefined user-interface code of just a few devices is unfair.

To determine the relationship between the functionality and space cost of a generator, we implemented a much simpler GUI generator for Java objects. It extracts the methods of a Java object and creates a frame consisting of a button for each method in alphanumeric order. It does not support customization, state presentation, menus, and many other features in `ObjectEditor`. The generator consumes 9621 bytes, which is even less than the space consumed by the code of the receiver’s command-only GUI.

Device	Space Consumed (bytes)		
	GUIs		SUIs (predefined)
	Remote-Control-like	Remote-Control-like with 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

Table 4. Amount of space consumed by the code of each device' handcrafted user-interface.

The SUI generator also portrays a different picture of generation costs. It consumes 20506 bytes, which is less than the space needed for the six devices' handcrafted SUIs. In general, the generation approach should represent a point between the two extremes of the client-factory and remote-factory approaches. It requires clients to have enough space to store a user-interface generator, but this space should be much less than the space needed to store user-interface code for all devices a user might ever want to use.

3.5.2 Deployment Time Costs

Qualitatively, the client-factory approach should have a lower user-interface deployment time than the remote-factory approach. The reason is that there is generally less delay in retrieving and executing user-interface code from local storage than from a remote source. The approach should also have a lower deployment time than the generation approach because it avoids downloading a device's description and then creating a new user-interface at interaction time.

We cannot make such qualitative arguments about the generation and remote-factory approaches. The reason is based on two ideas:

- 1) For generation, the times are highly dependent on processor speeds since the process requires executing a complex algorithm.
- 2) For factory downloading, the times are highly dependent on network speeds since the process requires retrieving all user-interface code from a remote machine.

As a result, we performed experiments to gather quantitative data. In particular, we measured the times for: (1) locally generating user-interfaces and (2) downloading and executing user-interface code from our department's web-server. To quantify the time benefits of the client-factory, we also measured the time it takes for clients to locally load and execute code.

To deploy the user-interfaces of the six networked devices, we used two different kinds of clients: (1) a laptop {Windows 2000 OS, 733 MHz Pentium, 128MB} and (2) an Ipaq pocket PC {Savage Java-based operating system [31], 206MHz StrongArm, 32MB}. We chose a pocket PC and a laptop as our experimental clients because they are mobile computers with a significant difference in processing power. This difference allowed us to investigate how a client's processing power can effect deployment time. Another factor that can effect deployment time comes from the fact that all of our generators and handcrafted were written in Java. Since the laptop, unlike the Ipaq, is not inherently Java-based, it must first start a Java Virtual Machine (JVM) before executing any deployment code. This process could increase total deployment times. To measure the possible increase, we also compared the times of deploying user-interfaces with and without a preloaded JVM. We preloaded JVMs by launching our user-interface deployment code from already running Java programs. These programs do not perform any computation that can effect deployment time. They simply launch our user-interface deployment code.

As implied above, network speeds can effect deployment time. Our evaluation thus includes a comparison of deployment times using different network speeds. We particularly consider the speeds of dialup, wireless, and wired LAN network connections.

Yet another factor that could effect deployment time is the type of user-interface deployed. Thus, we compared the times of generating GUIs and SUI using the generators mentioned above. We also compared the times of deploying user-interfaces with and without state. Incorporating state can effect deployment time since it requires executing code to render state widget (or views). It also requires downloading current property values in order to initialize state widgets (Figure 27).

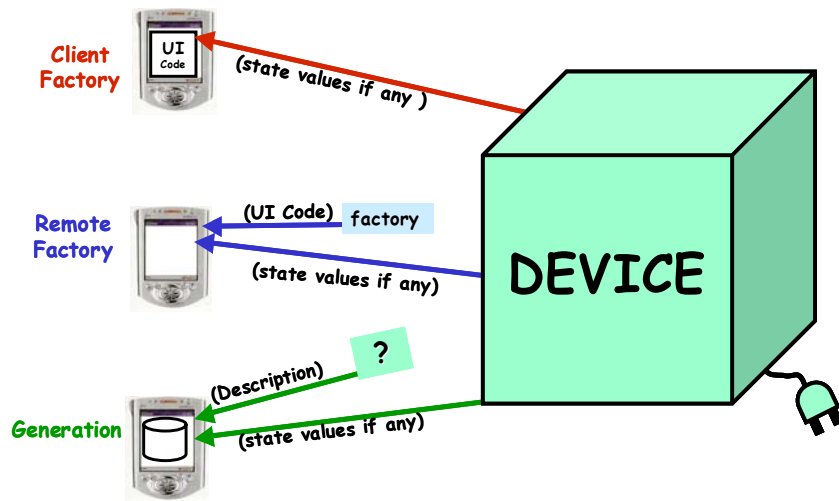


Figure 27. The downloaded components of the factory and generation approaches.

Finally, by measuring the times for deploying user-interfaces for our six devices, we can evaluate the effect of device complexity on deployment time. Consider the two extremes of device complexity—the lamp and the receiver. The lamp only has four commands and two state properties while the receiver has forty-two commands and twenty-two properties. Logically, the receiver should have a longer deployment time than the lamp.

3.5.2.1 Deployment Times on Laptop

Using a wired LAN connection on the laptop, we performed ten trails of collecting deployment times with and without a preloaded JVM. Under no-preloaded-JVM deployment, all the approaches yield significantly high times. Consider the task of deploying a command-and-state based GUI for the receiver. It takes approximately 8 and 9 seconds to deploy the handcrafted GUI (Figure 24a) using the client and remote-factory approaches respectively. There is a very small proportional difference (1.12) between the client and remote-factory approach. Such a small number is likely due to the fast network we used, which provided fetch times that were comparable to the laptop’s own disk access time. Generating the user-interface takes over 17 seconds, which is approximately twice the times of the factory approaches (Figure 24b).

In general, the times are much lower under preloaded-JVM deployment. The client and remote-factory times drop down to approximately 396 and 401 milliseconds (ms) respectively—under half a second. Again, there is also a small proportional difference (1.01) between the two approaches. Generating the receiver GUI reduces to approximately 7.5 seconds. This time is considerably lower than the generation times under no-preloaded-JVM deployment. However, it is still very high when compared to the preloaded-JVM factory times. We hypothesized that a considerable part of it is due to the process of starting up the generator and loading it into memory. Thus, we explored the idea of running the generator ‘in the background’, which means that the client keeps the generator always loaded and ready in memory. This general idea of running applications ‘in the background’ to avoid long startup times is not new. It has even been used by popular applications such as Netscape’s web-browser. Given our Java-based implementation, a generator loaded in memory assumes that a JVM is also running. It does not make sense to consider the case of always keeping predefined user-interface programs in memory since they are not universal applications. That is, functionality (or code) from one predefined user-interface cannot be used to help deploy another. A generator, on the other hand, is universal because all user-interfaces share the same generation algorithm.

Our results show that applying the idea significantly lowers generation time. The receiver GUI generation time actually drops down to approximately 882 ms. This time, however, is still over twice the times of the factory approaches under preloaded-JVM deployment. Figures 28 and 29 respectively present times for deploying command-only and command-and-state based GUIs for all six devices using the same laptop and 100 Mbps network connection.

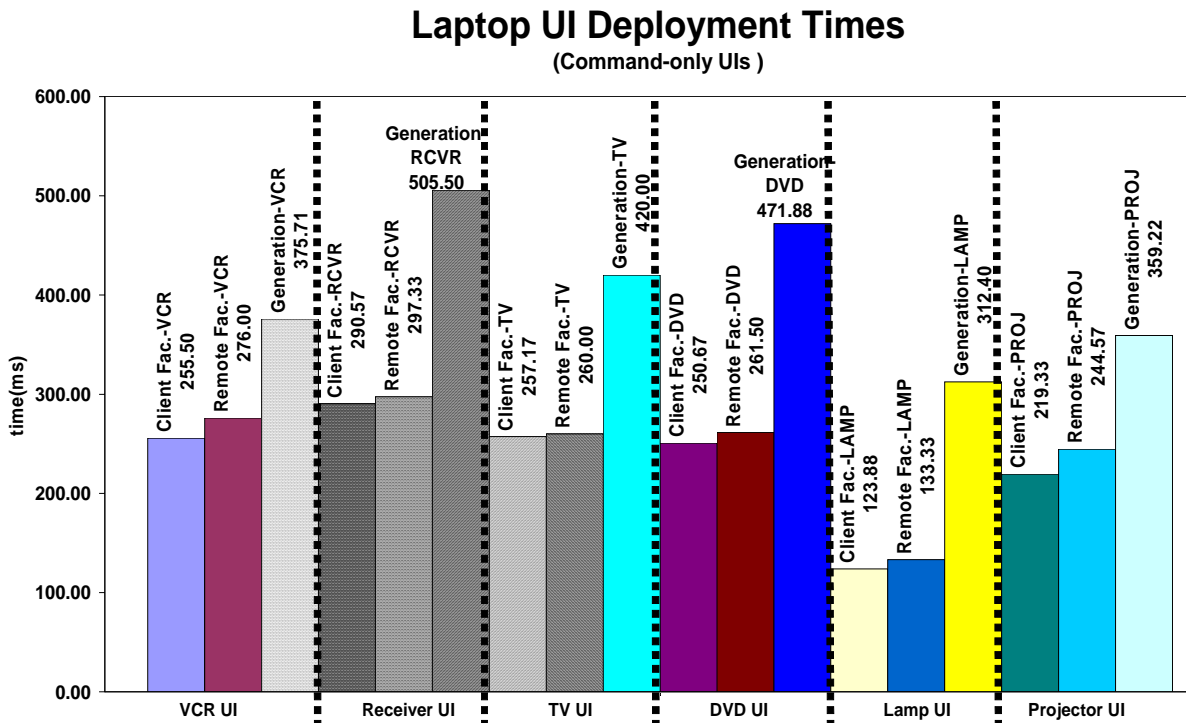


Figure 28. Command-only GUI deployment times for all six devices (using the laptop, ObjectEditor preloaded in memory, and a wired LAN connection).

The graphs show that the trend described above is not specific to just the receiver. In fact, for all of the devices we networked and the two kinds of GUIs, there is a considerable proportional difference between the deployment times of the generation and factory approaches. Generating the DVD player's command-only user-interface is just under twice as long as locally loading code. Generating its command-and-state based GUI is over 2.5 times longer than locally loading code. Thus, including state apparently creates a wider gap between the two approaches. When moving from command-only to command-and-state based GUIs, deployment time increases at a greater rate in the case of the generation approach. The DVD player's command-and-state based GUI takes nearly 1.5 times longer to generate than the command-only GUI. Under the client-factory approach, however, there is only a 1.08 difference in deploying the two kinds of use-interfaces. This greater increase in the generation approach quantifies the additional burden of creating state-based property displays 'on the fly' versus creating them from predefined code. Recall that ObjectEditor generates a device's state display panel by first using reflection to extract state property information. It then dynamically maps the

properties to appropriate widgets for displaying their values. The code for the predefined GUIs, however, avoids the need to perform reflection and dynamic mapping during interaction time. The GUIs are device-specific, and thus we could ‘hardwire’ the mappings between properties and widgets in their code.

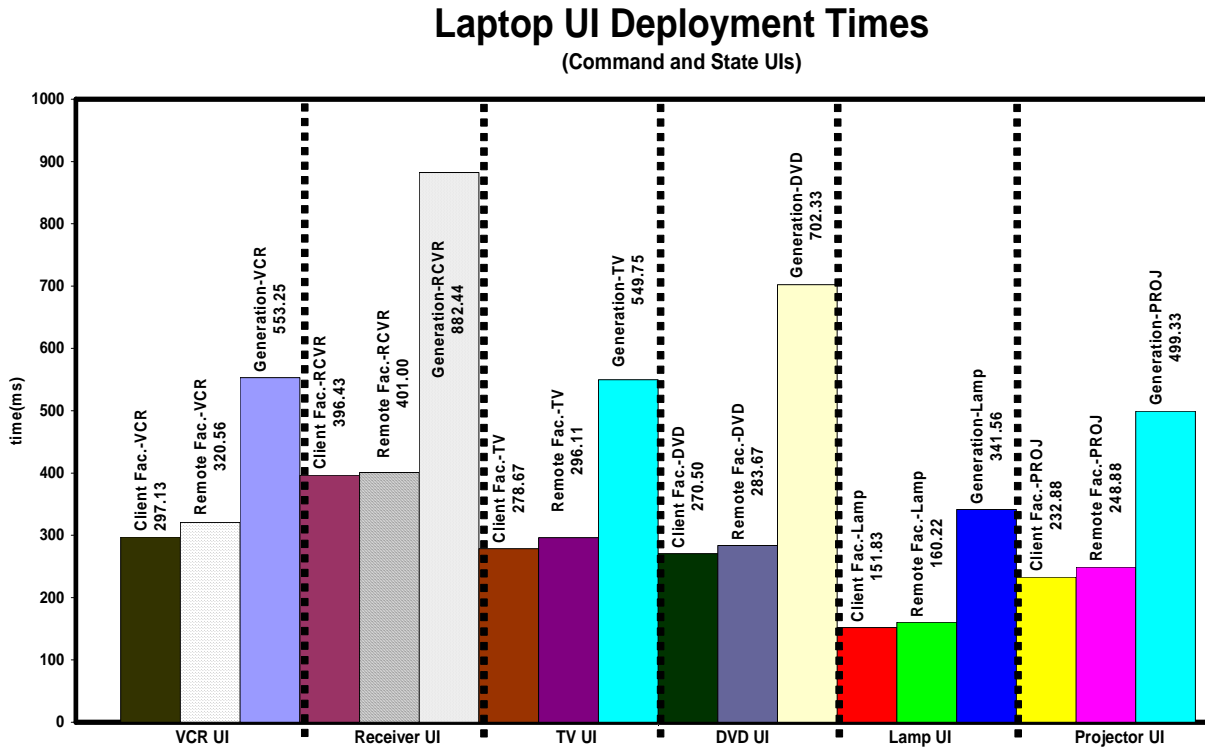


Figure 29. Command-and-state based GUI deployment times for all six devices (using the laptop, ObjectEditor preloaded in memory, and a wired LAN connection).

The graphs also show that deployment time can significantly increase with device complexity. Recall the two complexity extremes, the lamp and receiver. For all three approaches (generation, remote-factory, and client-factory), the receiver’s command-and-state based GUI takes over twice the time to deploy than that of the lamp. With the command-only receiver GUI, the generation time is 1.6 times greater than that of the lamp. The client and remote factory times for the receiver’s command-only GUI are over twice that of the lamps. Considering devices of more comparable complexity, like the TV (25 commands and 9 properties) and projector (23 command and 4 properties), the deployment times have smaller differences. The TV’s command-and-state based GUI deployment times under the three approaches are under 1.2 times longer than those of the projector.

For the rest of this dissertation, we assume that (a) the all deployment times are from a preloaded-JVM deployment and (b) the generation times are gathered from a preloaded generator. Further, all the deployment time values we present are averages of ten trials.

3.5.2.2 Deployment Times on Ipaq

Given that the Ipaq has a much slower processor than the laptop, it should offer considerably longer deployment times. We collected GUI deployment times for three devices (the lamp, projector, and receiver) on the Ipaq using a wired LAN network connection. The times (Figure 30) have a trend that is similar to the laptop results. The client-factory approach is the fastest. It is followed by the remote-factory approach, which is 1.44 times slower than the local factory. Generation is the slowest approach—taking approximately 2.8 and 1.9 times longer than the client and remote-factory approaches respectively.

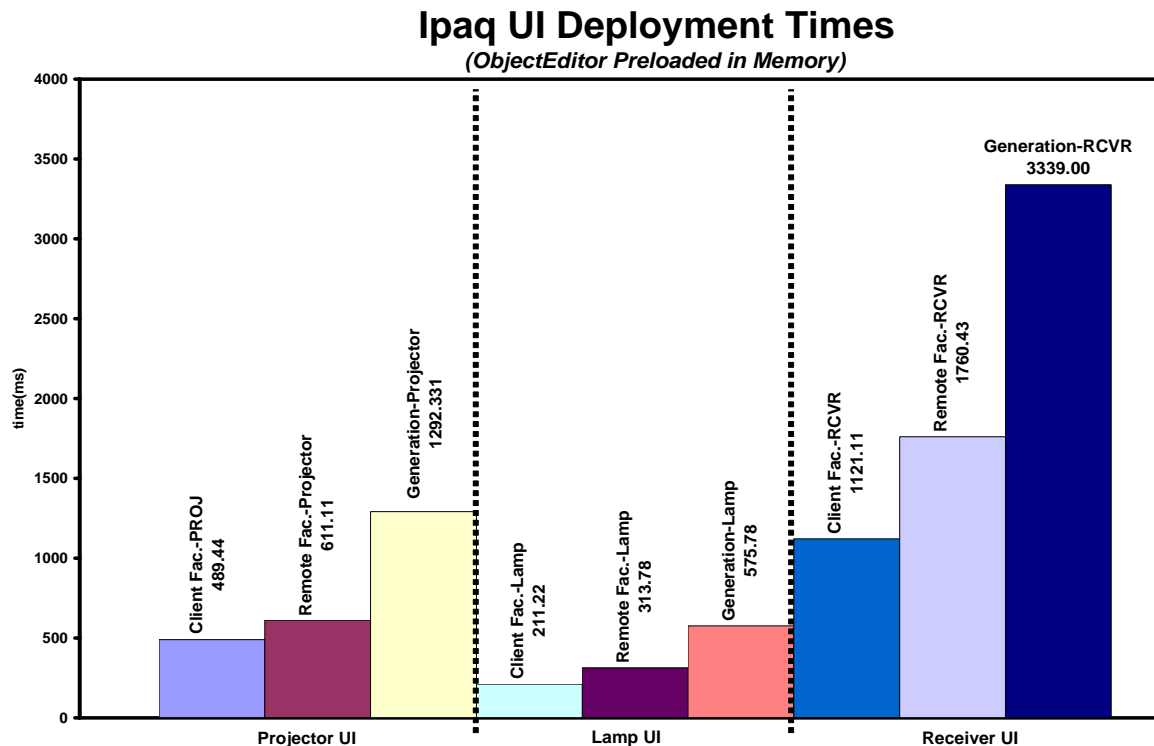


Figure 30. Command-and-state based GUI deployment times for the projector, lamp, and receiver (using the Ipaq and a wired LAN connection).

These Ipaq results quantitatively show a direct relationship between computation power and deployment time. The Ipaq, which is much less powerful than the laptop, yields significantly higher deployment times under the same scenarios (Figure 31). Consider the task of deploying a command-and-state based receiver GUI. The client-factory approach takes 1121.11 ms on the Ipaq versus 396 ms on the laptop. On the Ipaq, the remote-factory approach takes 1760.43 ms, but it takes 401 ms on the laptop. Finally, generation takes 3339 ms on the Ipaq as opposed to 882 ms on the laptop. The generation, remote-factory, and client-factory approaches respectively take 3.78, 4.4, and 2.8 times longer on the Ipaq than on the laptop. It even takes less time to generate the GUI on the laptop than to deploy one on Ipaq using a local factory.

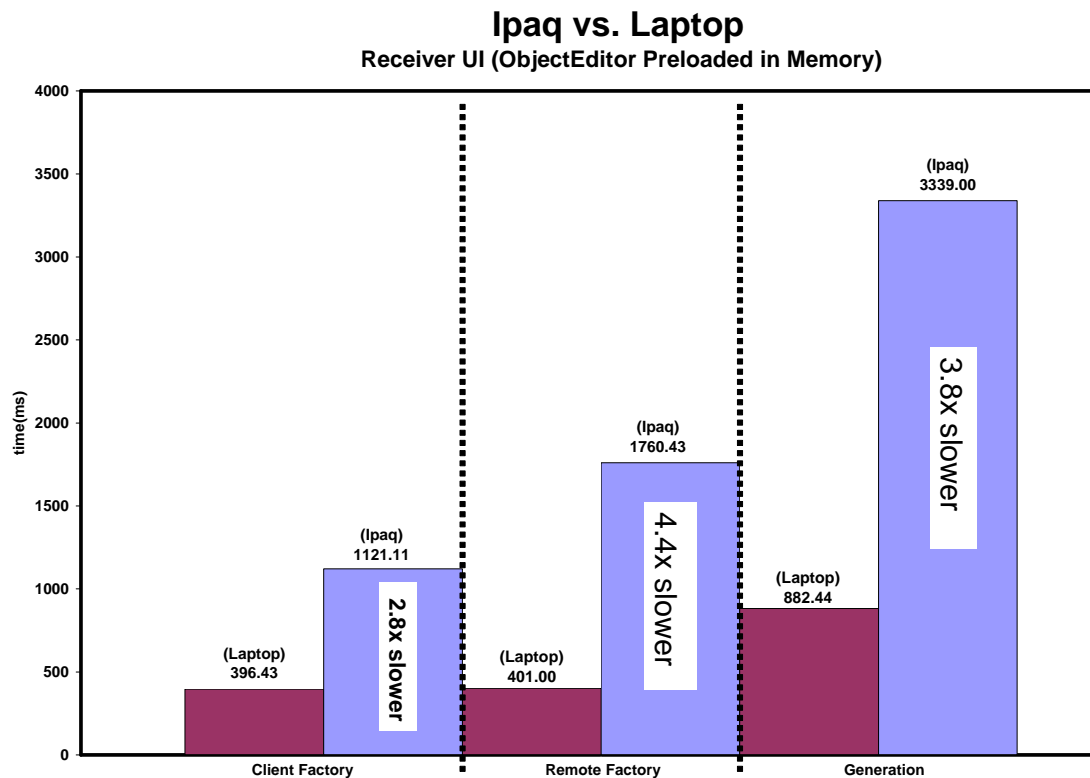


Figure 31. A visual display of the significant differences between Ipaq and laptop GUI deployment

3.5.2.3 Deployment Times Using Different Network Speeds

As Figure 27 shows, the factory and generation approaches have different levels of network dependency. For example, the client-factory approach is less network dependent than the remote-factory approach because it involves loading local user-interface code, instead of downloading. To discover the effects of different network speeds, we deployed the command-and-state based receiver GUIs on the laptop using a dialup (50Kbps) and wireless (1Mbps) network connection. The computer science department provides a dialup and wireless network for students and faculty, thus we were able to also run these experiments inside the building. The wireless card that we used in the experiments is capable of speeds up to 11Mbps. However, actual tests showed that our wireless network's bandwidth is actually much below its peak. In the following discussion, we assume that customization code for the generator is located locally on the client and is thus not downloaded.

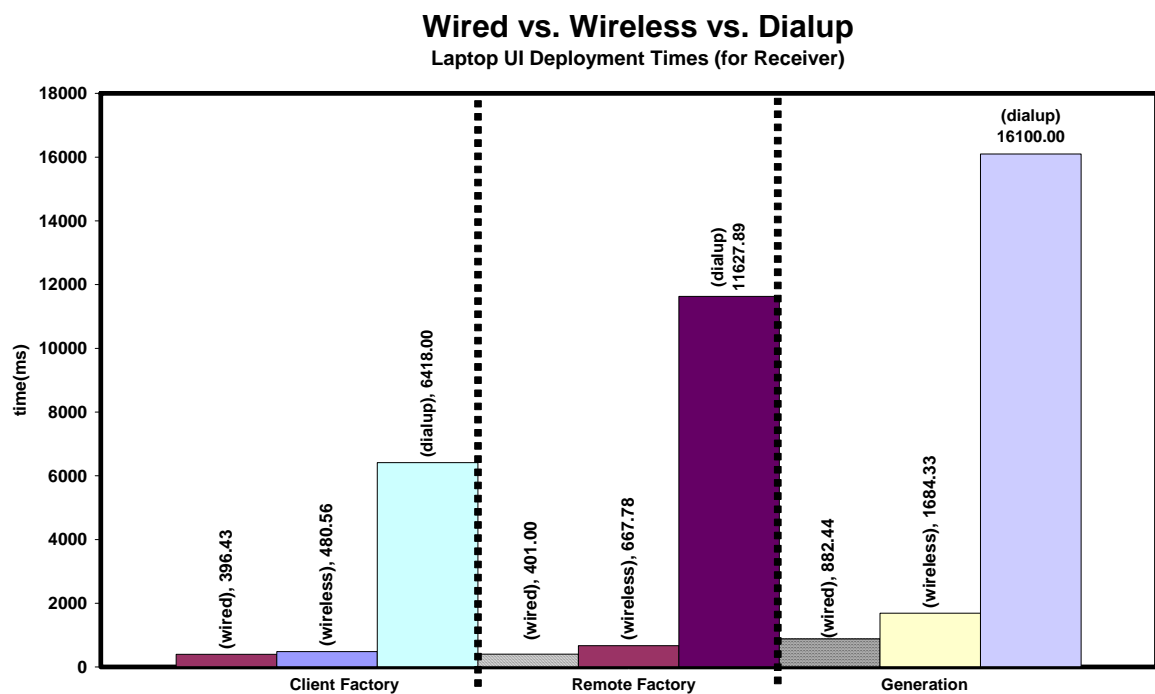


Figure 32. Command and state based receiver GUI deployment times using the laptop and different network speeds.

Our results show that as network speeds decrease, deployment time increases for all approaches (Figure 32). Using the client-factory approach, the wireless and dialup connections respectively take 1.21 and 16.19 times longer than the wired LAN

connection. The client-factory approach requires downloading from a network only when initializing the property values in a user-interface displaying device state. Since the above client-factory results are from deploying such user-interfaces, they show how the cost of filling in state widgets with current property values thus grows as speeds decline.

The proportional differences are even greater under the remote-factory approach because it is more network-dependent. Under the approach, a client 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 client-factory approach. Before downloading property values, a generator must first download device descriptions from which it creates user-interfaces. In our implementation, descriptions are the programming interface class of a device. Using generation, the wireless connection is 1.9 times slower than the wired LAN times. In essence, this result implies that 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 100 Mbps connection.

The results also indicate that decreasing the network speed creates a greater gap between the deployment times of the two factory approaches. Recall that with the wired LAN connection, deploying the command-and-state based receiver user-interface under the client-factory approach was only 1.01 times faster than the remote-factory approach. With the wireless connection, the remote-factory approach takes 1.39 times longer than the client-factory approach. The gap is even greater when using the dialup connection. Here, the remote-factory approach takes 1.81 times longer than the client-factory approach. A logical explanation for these greater proportional differences, as implied by the above discussion, is based on the fact that the remote-factory approach is more network-dependent. Therefore, as network speeds drop, the deployment time of the remote-factory approach increases at a faster rate than when using a local factory.

3.5.2.4 SUI Deployment Times

Using the laptop and a wired LAN connection, we collected SUI deployment times for the lamp, TV, and receiver. The results (Figure 33) indicate that SUI deployment follows the same trend as GUI deployment. That is, the client-factory approach is fastest, followed by remote factory, and then generation. Also, as device complexity increases, the deployment time of each approach increases.

In the SUI case, however, the differences between the three approaches were not as significant as they were with GUIs. The remote-factory approach is only 1.03 times slower than the client-factory approach and 1.14 times faster than generation. Moreover, generation is only 1.18 times slower than the client-factory approach.

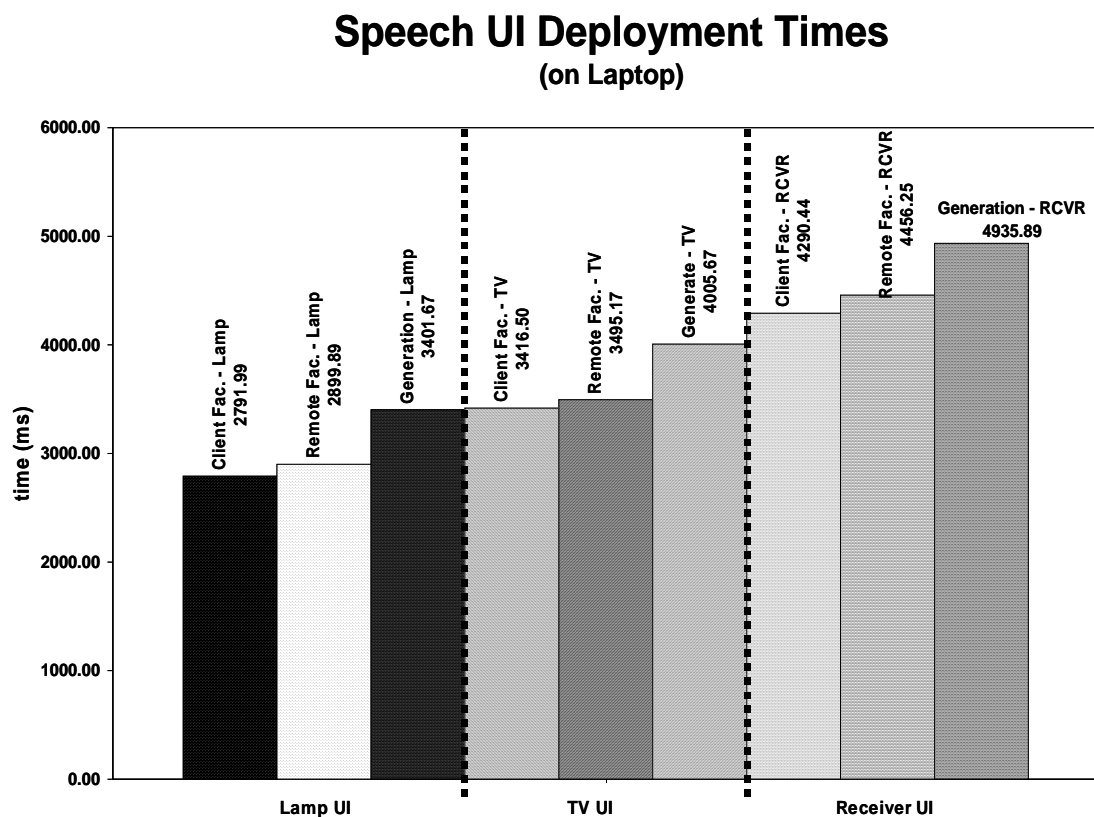


Figure 33. Command-only SUI deployment times for the projector, lamp, and receiver (using the Laptop and a wired LAN connection).

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 client-factory 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.

3.5.3 Operation Invocation Time Costs

The indirection in creating a user-interface ‘on the fly’ implies that the generation approach should yield longer operation invocation times than the predefined approach. We can illustrate this indirection by comparing `ObjectEditor`’s approach to handling button push events to that of the predefined GUIs. Recall that regardless of the GUI’s source, a button push should lead to invoking the device object’s method that shares the button’s name.

After a push event, `ObjectEditor` executes a method described by following pseudocode:

```
HandleButtonPush(Object t, String button_name)
    Method m = findMethodFromButtonName(button_name, t);
    t.invoke(m);
}
```

Given the name of the pushed button and a reference to a target device object, `handlebuttonpush()` first dynamically finds the `Method` object that is associated with the button name. It then uses reflection to request the device to invoke the actual method encapsulated this `Method` object. As the code shows, `handlebuttonpush()` is generic. That is, it was written to process push events on any button generated for any device. This generality is a requirement since `ObjectEditor` creates GUIs for arbitrary devices. A consequence of this generality is that `handlebuttonpush()` cannot directly reference methods of a specific device—hence the need for a dynamic search (i.e. execute `findMethodFromButtonName()`). The predefined approach, however, allows us to write user-interface programs for specific devices. We could thus directly reference the devices’ methods in code and avoid the need for dynamic searches and reflection-based method invocation.

Even with this indirection, the mean operation invocation times of the six generated user-interfaces are insignificantly higher than that of the handcrafted predefined user-interfaces. On average, there was just a 0.5% difference. The predefined approaches (client and remote-factory) deployed the same user-interface, just from different sources. As a result, they have the same operation invocation times.

3.6 *Device Binding Time*

There are two times a client can learn about (or bind to) a device so that it can deploy a 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 client-factory approach inherently supports early binding, and the remote-factory and generation approaches support late binding.

3.7 *Deployment Reliability*

Outside of binding time issues, there may be other cases in which an approach is unable to deploy a user-interface for a given device—even for a properly functioning client. This notion is particularly true for the remote factory and remote generation approaches, which involve using mechanisms executing outside of a client in order to deploy a user-interface. Both approaches require accessing a factory or generator on a machine that could be overwhelmed with requests from multiple clients. A device, for example, could be so busy processing remote commands from users that it is unable to handle user-interface requests.

In the third-party-factory and remote generation approaches, a client must explicitly make a network connection to a machine that is not the target device. This dependency on an additional network connection makes the two approaches more vulnerable to network problems than other approaches. The client-factory and client-side generation are capable of deploying user-interfaces without the need of an external machine thus

they offer more reliability than the other approaches. We leave a quantitative evaluation of deployment reliability as future work.

3.8 Conclusion

In this chapter, we identified several dimensions along which existing user-interface deployment approaches can be compared. Using these dimensions, we qualitatively and quantitatively evaluated the various approaches. The evaluation presents several important results.

Within the domain of remote-control-based user-interfaces, it shows how the user-interface flexibility of the predefined approach is a little greater than that of the generation approach. The SUI generator can fully automatically create SUIs that are identical to the remote-control-based ones we handcrafted. However, even with the support of declarations, *ObjectEditor* is unable to semi-automatically generate one aspect of the remote-control-based GUIs—placement of new labels.

The evaluation also compares the programming costs involved in these different approaches. Fully automatic generation has a one-time programming cost of writing a generator. With the predefined approach, however, programmers must manually write user-interfaces for each kind of device and toolkit. Our data shows that the amount of code needed to handcraft user-interfaces can be relatively small. However, the process requires a programmer to actually spend time doing it. As a result, the predefined approach does not always guarantee a user-interface for any given device. Fully automatic generation, on the other hand, has a zero declaration cost and thus guarantees user-interfaces.

Our maintenance costs evaluation shows further benefits of the generation approach. The approach is the easiest to change to support new user-interface toolkits—it only requires writing a new generator. On the other hand, the predefined approach requires writing new user-interfaces code for a possibly unlimited number of devices. Another aspect of maintenance costs is the amount of work involved in updating a deployment infrastructure as devices removed from a network. The client-factory and third-party-

factory approaches require tracking and removing user-interface code because it is separated from devices. The fully automatic generation approach has the advantage of allowing the removal of devices without the need to remove any user-interface code. Semi-automatic generation, however, requires per-device user-interface code. It is no better, in this respect, than the client and third-party-factory approaches.

In our efficiency evaluation, we compared the approaches using several costs: device space, client space, deployment time, and operation invocation time. The generation, client-factory, and third-party-factory approaches consume the least amount of device space since they require no user-interface code on the device. The device-factory approach consumes the most device space. As mentioned earlier, this disadvantage is important due to the expectation that some networked devices will offer little storage. An advantage of storing user-interface code on devices, though, is low client space costs. The remote-factory and remote generation approaches also share low client space costs since the respective user-interface and generation code are on separate machines. In comparing the client-factory and client-side generation approaches, the client space costs depend on the number of devices with which a user will interact and the complexity of the generator used. For our six devices, the space costs of the handcrafted GUIs were extremely lower than ObjectEditor due to the generator's complexity. The SUI generator, on the other hand, consumes less space than the six handcrafted SUIs.

Our deployment time evaluation presents numerous results. Under all experiments, the client-factory approach is the fastest. The remote-factory approach and then generation follow it. The generation times are particularly important because they are generally multiple times longer than those of the factory approaches. There are some qualities shared by all three approaches. Namely, deployment time significantly increases under several practical cases: (a) a JVM is not preloaded, (b) reduced client computation power, (c) deploying a command-and-state-based user-interface instead of a command-only one, (d) dropping network speeds, (e) increasing device complexity, and (f) deploying SUIs instead of GUIs.

Unlike deployment time, the operation invocation time metric has no impact in dividing the approaches. The last two metrics we discussed in this chapter (device binding time and deployment reliability) do yield important conclusions. One is that the client-factory approach supports early binding. Thus, it requires users to anticipate each device they will want to use and pre-install the appropriate user-interface code. This burden is high for users who wish to arbitrarily interact with any device. The generation and other factory approaches support late binding and thus avoid this particular problem. However, we show that the remote-generation, third-party-factory, and device-factory approaches are still susceptible to issues that can make them unreliable.

From our evaluation we found that each approach has a unique benefit—thus proving our *Uniqueness Hypothesis*. The predefined approaches (client and remote factory) all share the highest user-interface flexibility benefit of all the approaches. The client-factory separates itself by offering the lowest deployment time and highest reliability. On the other hand, the remote factory differentiates itself by offering the latest binding time and lowest client storage. Within the remote factory approach, the device-factory deployment has the lowest maintenance while third-party-factory deployment has the lowest device storage costs. For all generation approaches, the benefits are lowest programming costs and latest binding time. Client-side generation has a higher reliability than remote generation. However, it requires more client storage. Finally, fully automatic generation has lower maintenance costs than the semi-automatic approach, but its user-interface flexibility is not as good.

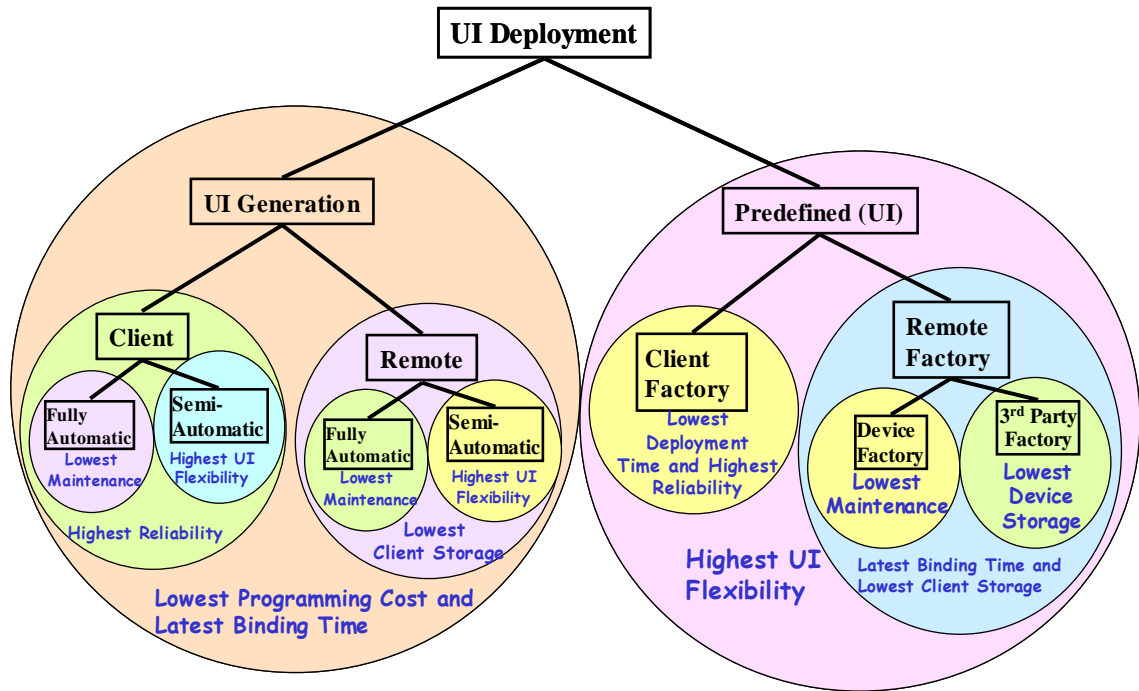


Figure 34. A graphical representation of the unique benefit(s) of each approach.

Figure 34 graphically depicts these unique benefits. To discover the unique benefit of an approach, simply perform a logical AND operation on the labeled benefit of each circle enclosing it.

Chapter 4: User-Interface Retargeting

Our evaluation in the previous chapter presents several limitations of existing deployment approaches. One important limitation is that it generally takes a long time to generate a user-interface. In particular, our GUI and SUI generators often take multiple times longer to deploy a user-interface than the client factory approach. These differences are drastically greater than the 100 ms long period of human noticeable delay [34]. From corresponding with the builders of the CMU GUI generator, we found that their system also has a significantly long generation time. They stated that on a PocketPC with a similar computation power as the Ipaq, it takes nearly 20 to 30 seconds to generate a GUI for an automobile (a GMC Yukon-Denali). The latency of their system affirms that the problem of long deployment times is a general characteristic of the generation approach.

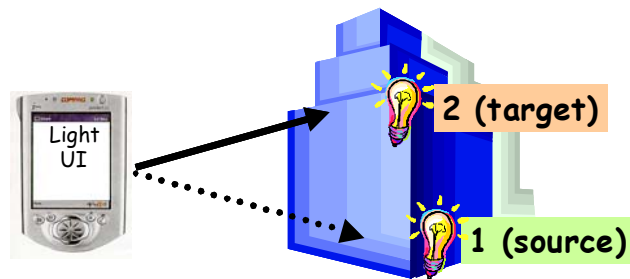


Figure 35. Retargeting a UI between two lights on different floors.

In this chapter, we address this issue of long generation times. Specifically, we prove the *Time-Efficient Generator Hypothesis*: it is possible for SUI and GUI generators to have deployment times that are often as good as or noticeably better than the inherently fastest approach of locally loading device-specific user-interface code. Our approach is based on the idea of user-interface retargeting, which involves dynamically mapping a previously generated user-interface of a (source) device to another (target) device that can share the user-interface. It could allow, for example, a security guard patrolling through a building to use the same generated user-interface for a hallway light to control other lights on different floors (Figure 35). The goal of retargeting is to recycle widgets. By

recycling parts of a previously generated user-interface of a device that a user is not using, a generator can significantly speed up the creation of a user-interface.

In the next section, we provide a more detailed overview of the idea and present the important limitations that currently exist in supporting it. We then describe the implementations of new retargeting mechanisms we built to overcome these limitations. Using the mechanisms, we evaluate how well retargeting allows us to prove the *Time-Efficient Generator Hypothesis*. Finally, we present our conclusions.

4.1 Overview

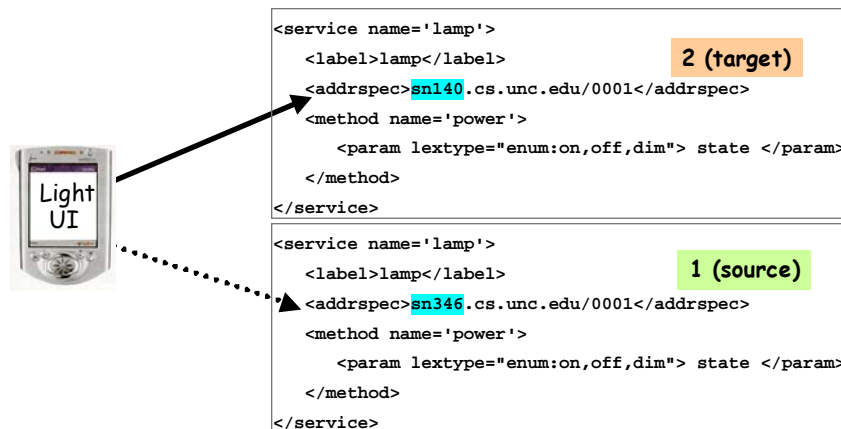


Figure 36. Retargeting between two lights in different rooms under Hodes' System.

The basic idea of device user-interface retargeting was previously identified and implemented in Hodes' System. The system retargets the user-interface of a source device to a target device when the XML descriptions of the two devices differ only in the device addresses (Figure 36). To retarget, the system simply changes the source user-interface's RPC address from the source device's address to the target device's address. In the example of Figure 36, the RPC address changes from `sn346.cs.unc.edu/0001` to `sn140.cs.unc.edu/0001`. The buttons and state widgets of the source user-interface stay the same throughout this process. However, the command invocations and state updates become associated with the target device. Hodes' System has source user-interface and programming interface flexibility limitations, some of which we overcome.

Source user-interface flexibility determines the kinds of user-interfaces a system can retarget. There are various levels of source user-interface flexibility a system can offer (Figure 37). A system could retarget user-interfaces containing operations possibly with arguments of primitive and/or structured types (Figure 38a). In addition, it could retarget user-interfaces possibly displaying primitive and/or structured typed properties (Figure 38(b and c)). Hodes' System can retarget command and state-based user-interfaces. It only supports primitive-typed command arguments and state.

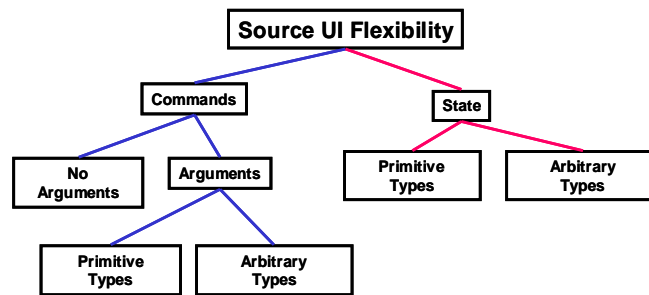


Figure 37. Levels of source UI flexibility.

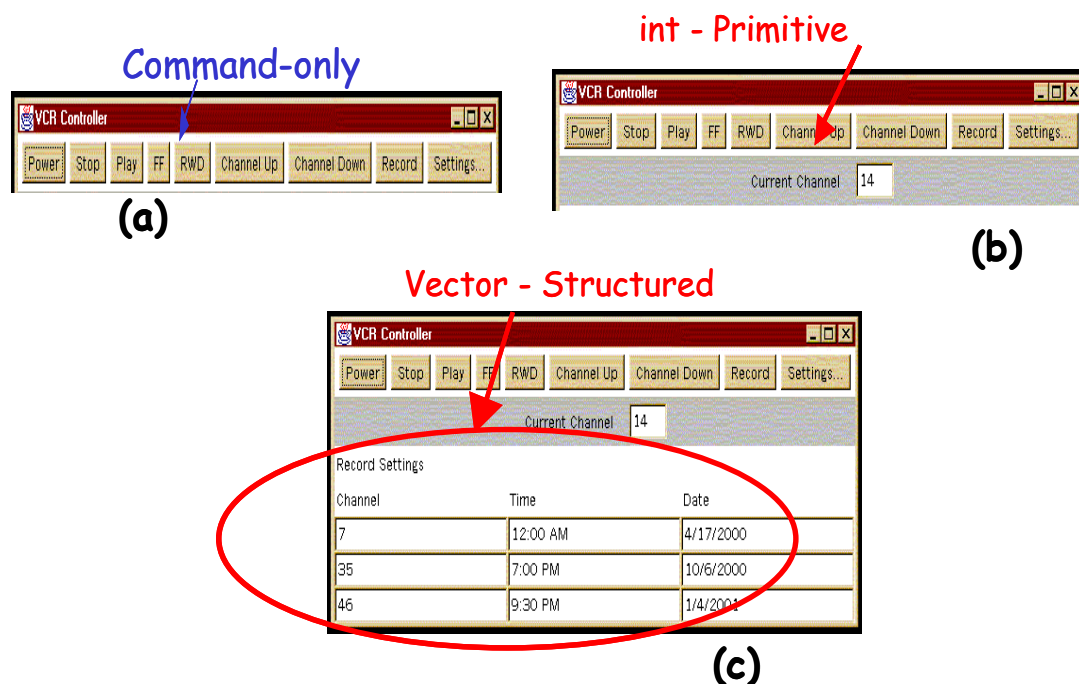


Figure 38. User-interfaces requiring different levels of source user-interface flexibility.

Programming interface flexibility determines how different the programming interfaces of a source and target device can be in order to support retargeting. As

supported by Hodes' System, a generator could retarget only if the two devices share the same programming interface. By sharing the same programming interface, they inherently share identical commands and state properties. Thus, when retargeting, no time must be spent changing the appearance of the source user-interface to fit a target device (Figure 39). Time must only be spent changing the code associated with the components to direct method invocations to the target device and reflect the device's state changes.

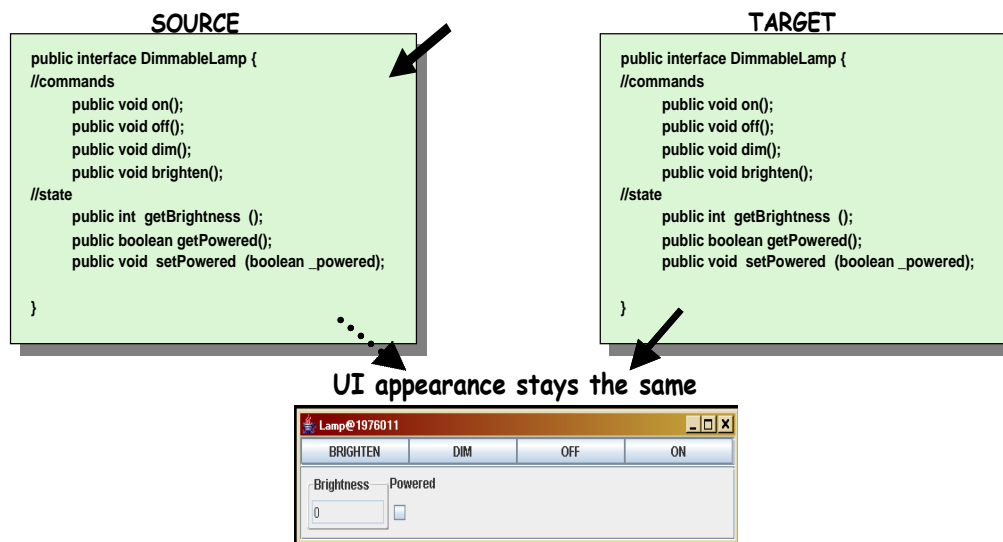


Figure 39. Devices with identical programming interfaces share identical user-interfaces.

There is an important limitation with this level of retargeting. It can only be used if a user has recently interacted with a device of the same programming interface as the target device. Otherwise, the generator must spend the time creating a whole new user-interface. In this case, rather than create a new user-interface, a generator could change the source device's user-interface to fit the target device, recycling parts of the user-interface that the two devices share. For example, a generator could change the user-interface of a non-dimmable lamp to fit a dimmable lamp (Figure 40). Conversely, the generator could retarget the dimmable lamp's user-interface to the non-dimmable lamp. This task would involve removing dim-related components from the user-interface.

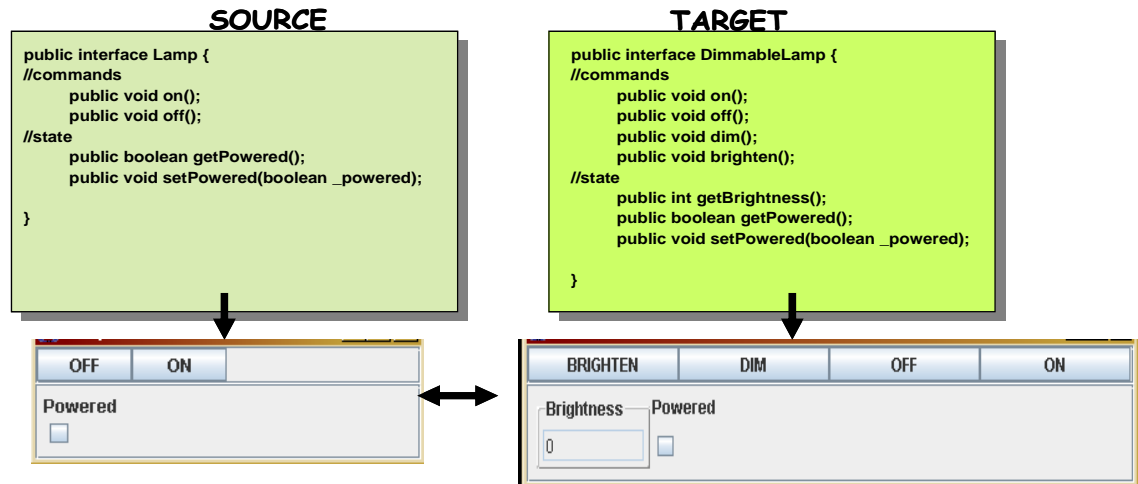


Figure 40. Retargeting between a dimmable and non-dimmable lamp.

Retargeting when the source and target devices have different programming interfaces does not always require changing the source user-interface. Figure 41 shows an example of this case. The ‘Current Channel’ properties of two different VCRs, though represented by different types (String and int), map to the same views. This is also true for the ‘Record Settings’ properties. The ‘Current Channel’ maps to a textbox for entering integers or strings and the ‘Record Settings’ maps to a special widget for displaying the elements of vectors or arrays.

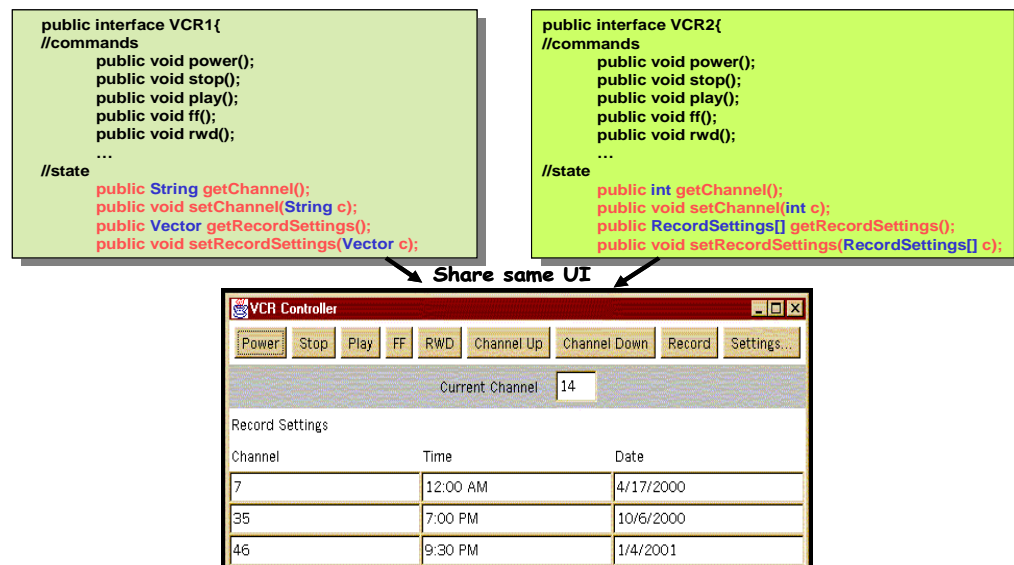


Figure 41. Two different VCR programming interfaces that can share the same UI.

Beyond its flexibility limitations, Hodes’ work does not provide any quantitative evaluations showing the deployment time benefits of retargeting. It also supports

retargeting only in the domain of GUIs. As Table 5 shows, our goal was to offer more GUI retargeting flexibility and additionally support SUIs. We also wanted to provide a quantitative evaluation of retargeting.

We based our GUI retargeting support on the ObjectEditor framework. The generator provides mechanisms, unsupported by our SUI generator, which allowed us to achieve higher flexibility goals. For example, it can create state-based user-interfaces, thus allowing us to retarget such user-interfaces. The SUI generator, however, does not support state. We did implement a basic level of retargeting for this generator to quantify any possible benefits of retargeting SUIs.

				Hodes Retargeting	SUI Gen. Retargeting	Object Editor Retargeting
Source UI Flexibility	Command UIs	Arguments	Primitive Types	Y	N	Y
			Structured Types	N	N	Y
		No Arguments			Y	Y
	State UIs	Primitive Types		Y	N	Y
		Structured Types		N	N	N
Programming Interface Flexibility	Different Programming Interfaces	Identical UIs		N	N	Y
		Non-Identical UIs		N	N	Y
	Same Programming Interfaces			Y	Y	Y

Table 5. Retargeting flexibility – Hodes’ System vs. Our Goals.

4.2 GUI Retargeting

To retarget a GUI, ObjectEditor uses the source and target devices’ programming interfaces to find the names of:

- Target-Only Commands (TOC): the target device commands that the source device does not share
- Source-Only Commands (SOC): the source device commands that the target device does not share
- Shared Commands (SC): the commands that the two devices share
- Target-Only Properties (TOP): the target device properties that the source device does not share

- (e) Source-Only Properties (SOP): the source device properties that the target device does not share
- (f) Shared Properties (SP): properties that the two devices share.

Using this information, it retargets the user-interface using the algorithm described below:

Let TOC, SOC, SC, TOP, SOP, SP = lists corresponding to (a)-(f) above
 {We will expand on how these lists are computed later.}

Let T = the target device

Let U = the source user-interface object

```

retarget (T,U, TOC, SOC, SC, TOP, SOP, SP) {
    U.disable()
    for each command_name (a) in TOC {
        x = new Button(c);
        setButtonTarget(x,T);
        U.add(x);
    }
    for each command_name (b) in SOC {
        x = U.getButton(b)
        U.remove(x);
    }
    for each command_name (c) in SC {
        x = U.getButton(c)
        setButtonTarget(x,T);
    }
    for each property_name (d) in TOP {
        t = getType(T,d);
        x = getMatchingWidget(t);
        setLabelandOtherAttributes(x);
        setWidgetTarget(x,T);
        U.add(x)
        updateWidget(x);
    }
    for each property_name (e) in SOP {
        x = U.getWidget(e)
        U.remove(x);
    }
    for each property_name (f) in SP {
        x = U.getWidget(f)
        setWidgetTarget(T,x);
        updateWidget(x);
    }
    U.enable()
}

```

Given a reference to a source user-interface object, target device, and the retargeting lists (*TOC*, *SOC*, *SC*, *TOP*, *SOP*, and *SP*) the method creates a new appropriately labeled button for each target device command that the source device doesn't share. It maps the each button to the target device method of the same name (using `setButtonTarget()`) so that pushing the button invokes the corresponding method. Next, it adds each new button to the user-interface. The algorithm then removes the button of each source device command that the target device does not share. Then, it remaps the button of each command shared by the two devices to the target device method of the same name.

The algorithm follows a similar process for handling properties as it does with commands. For each target device property that the source device does not share, the algorithm creates a new appropriately labeled widget that can display values of its type. Recall from our discussion of `ObjectEditor` that the generator can automatically return such a widget given a property type. Our algorithm simply calls this code (i.e. `getMatchingWidget()`). It then calls `setWidgetTarget()` to associate each widget with its matching property's getter and setter method. This process ensures that `ObjectEditor` calls the appropriate getter method when retrieving values to display in the widget. It also guarantees that the generator calls the appropriate setter method to update the target device property when value changes are made on the corresponding widget. After this process, the algorithm adds the widget to the user-interface and initializes it with the current value of its associated property. Next, it removes the widget of each source device property that the target device does not share. Then, it associates the widget of each property that the two devices share to the target device's getter and setter method of the property. Each widget gets updated so that it shows the target device's value of the property.

To illustrate the algorithm, consider the scenario of retargeting a non-dimmable lamp GUI to a dimmable lamp (Figure 40). List A contains the names 'brighten' and 'dim'. The algorithm creates two new buttons labeled with these names and then maps them to the target device's `dim()` and `brighten()` methods, respectively. It then adds the two buttons onto the GUI. List B is empty since the target (dimmable) lamp's list of command names is a superset of the source (non-dimmable) lamp's corresponding list.

Consequently, the algorithm moves on to process list C, which contains the names ‘on’ and ‘off’. It maps the on and off buttons to the target lamp. List D contains the single name ‘brightness’. This is the name of the only property that the two lamps do not share. The algorithm creates a textbox labeled ‘brightness’ for displaying the integer-based value of the property. It associates the textbox to the target lamp’s `getBrightness()` and `setBrightness()` methods. Then, it adds the textbox to the source user-interface and initializes it with the integer value returned by `getBrightness()`. The algorithm skips the next step since list E is an empty—the target device offers the source device’s only property called ‘powered’. As a result, the ‘powered’ widget remains on the user-interface. However, it changes this widget’s associated getter and setter method respectively to the target lamp’s `getPowered()` and `setPowered()` methods.

The retargeting algorithm together with ObjectEditor’s inherent functionality, allows us to meet our GUI retargeting goals (Table 5). As the first half of the algorithm shows, our implementation retargets command-based user-interfaces by supporting the addition, removal, and remapping of buttons. Recall from our initial description of ObjectEditor (in 2.1.6) that if a command requires arguments of primitive or structured types, the generator creates a dialog box for entering desired parameter values. This feature automatically allows ObjectEditor to retarget user-interfaces with command arguments of primitive and structured types. The generator simply treats button pushes on retargeted user-interfaces in the same manner as fully generated ones. As Table 6 shows, this ability to support commands with structured typed arguments, when retargeting, allows our mechanism to offer more source user-interface flexibility than Hodes’ System.

The second half of the algorithm shows that our implementation also retargets state-based user-interfaces by supporting the addition, removal, and remapping of property widgets. It particularly supports primitive typed properties, which make up of all of the state of our experimental networked devices. It can expand and/or contract a source user-interface to fit a target device with different commands and properties than the source device. This ability raises two related and important issues we addressed in our implementation:

- (1) *Fastest User-Interface Selection*: For a target device, let us assume that a generator has two or more potential source user-interfaces loaded in memory and none of them was created for a device that is the same type as the target. How should the generator select a source user-interface that can be changed to fit the target device in the least amount of time? To illustrate this issue, imagine a person at work with a client that has the user-interfaces of some frequently used home devices still running. If this person wants to use a conference room projector and such a device does not exist at home, which source user-interface should the generator pick so that it spends the least time retargeting?
- (2) *Approach Selection*: How should a generator decide whether it is faster to retarget the ‘fastest’ user-interface or generate a new one for the target device?

We address both issues using the novel idea of *regression-based source-device prediction*. A generator selects the fastest user-interface to retarget by using a function that estimates the retargeting time of each potential source user-interface. This function accepts the amount of work required in changing a user-interface and returns a retargeting time estimate. Given an estimate for each potential source user-interface, the generator then selects the one with the lowest value. Similarly, to predict the faster of the two approaches (generate or retarget), a generator uses a function that estimates the time to create a new user-interface for the target device. This function accepts the amount of work required in creating a whole new user-interface and returns a generation time estimate. Given this generation time estimate and the retargeting time estimate for the fastest source user-interface, the generator selects the approach with the lowest value.

We derived an outline of the two estimation functions by first identifying the high-level steps (or sub-operations) involved in algorithms of the respective approaches. As our algorithm shows retargeting involves: (a) adding buttons for the target device commands that are not shared by the source device, (b) removing buttons of the source device commands that the target device does not offer, (c) remapping buttons of commands shared by the source and target devices to the target device, (d) adding widgets for the target device properties that are not shared by the source device, (e) removing widgets of the source device properties that the target device does not offer,

and (c) remapping widgets of properties shared by the source and target devices to the target device. Generation, on the other hand, involves: (a) creating an empty frame for the user interface, (b) adding buttons for invoking the target device's commands, and (c) adding widgets for displaying the target device's property values. As we will show later, the time it takes to remap, remove, and add a property widget depends on the type of widget. For example, the time it takes to create and add a string widget to a user-interface is more than the time it takes to perform the same operation on a boolean widget. We found three kinds of widgets with significant sub-operation time differences: number (int, float, double, and long), string, and boolean widgets. Thus appropriately, each approach's estimation function is the following sums:

$$1) T_{ret}(BA, BD, BR, NWA, BWA, SWA, NWD, BWD, SWD, NWR, BWR, SWR) =$$

$$T_{add_btn}(BA) + T_{rmv_btn}(BD) + T_{rmp_btn}(BR) + \\ T_{add_num_wdgt}(NWA) + T_{rmv_num_wdgt}(NWD) + T_{rmp_num_wdgt}(NWR) + \\ T_{add_bool_wdgt}(BWA) + T_{rmv_bool_wdgt}(BWD) + T_{rmp_bool_wdgt}(BWR) + \\ T_{add_str_wdgt}(SWA) + T_{rmv_str_wdgt}(SWD) + T_{rmp_str_wdgt}(SWR)$$

add; {BA=# buttons to add; NWA=# num widgets to add; BWA=# bool widgets to add; SWA=# string widgets to
NWD=# num widgets to delete; BWD=# bool widgets to delete; SWD=# string widgets to delete;
NWR=#num widgets to remap; BWR=# bool widgets to remap; SWR=# string widgets to remap}

$$2) T_{gen}(BG, NWG, BWG, SWG) = T_{gen_frm} + T_{gen_btn}(BG) + T_{num_pwdgt}(NWG) + T_{bool_pwdgt}(BWG) + \\ T_{str_pwdgt}(SWG)$$

{BG=# buttons to generate; NWG=# num widgets to generate; BWG=# bool widgets to generate;
SWG=# string widgets to generate}

T_{ret} estimates retargeting time by summing the results of functions that estimate the completion times of the retargeting sub-operations based on given workload values:

- $T_{add_btn}(BA)$ estimates the cost for adding BA buttons
- $T_{rmv_btn}(BD)$ estimates the cost for removing BD buttons
- $T_{rmp_btn}(BR)$ estimates the cost for remapping BR buttons
- $T_{add_num_wdgt}(NWA)$ estimates the cost for adding NWA number widgets
- $T_{rmv_num_wdgt}(NWD)$ estimates the cost for removing NWD number widgets
- $T_{rmp_num_wdgt}(NWR)$ estimates the cost for remapping NWR number widgets
- $T_{add_bool_wdgt}(BWA)$ estimates the cost for adding BWA boolean widgets
- $T_{rmv_bool_wdgt}(BWD)$ estimates the cost for removing BWD boolean widgets
- $T_{rmp_bool_wdgt}(BWR)$ estimates the cost for remapping BWR boolean widgets

Similarly, T_{gen} estimates generation time by summing the results of functions that estimate the completion times of the generation sub-operations based on given workload values:

- T_{gen_frm} estimates the cost for generating the (empty) enclosing frame on which the buttons and property widgets will be placed
- $T_{gen_btn}(BG)$ estimates the cost for generating BG buttons
- $T_{gen_num_wdgt}(NWG)$ estimates the cost for generating NWG number widgets
- $T_{gen_bool_wdgt}(BWG)$ estimates the cost for generating BWG boolean widgets
- $T_{gen_str_wdgt}(SWG)$ estimates the cost for generating SWG string widgets

Sub-operation T_{gen_frm} , in T_{gen} , has no parameters because generating an empty frame is a static operation—it should therefore have a constant value.

Given the outlines for the two time estimation functions, our next step was to define the actual calculations involved within the sub-operation functions. We achieved this goal by using regression, which is a method for deriving an empirical function from a set of experimental data. To gather the necessary empirical data, we used timestamps to measure actual times for performing the retargeting and generation sub-operations over a range of workloads. More formally, we profiled the generator to measure the actual times represented by the series below:

- $T_{add_btn}(1), T_{add_btn}(2), \dots T_{add_btn}(N), N=42$
- $T_{rmv_btn}(1), T_{rmv_btn}(2), \dots T_{rmv_btn}(N), N=42$
- $T_{rmp_btn}(1), T_{rmp_btn}(2), \dots T_{rmp_btn}(N), N=42$
- $T_{add_num_wdgt}(1), T_{add_num_wdgt}(2), \dots T_{add_num_wdgt}(N), N=16$
- $T_{rmv_num_wdgt}(1), T_{rmv_num_wdgt}(2), \dots T_{rmv_num_wdgt}(N), N=16$
- $T_{rmp_num_wdgt}(1), T_{rmp_num_wdgt}(2), \dots T_{rmp_num_wdgt}(N), N=16$
- $T_{add_bool_wdgt}(1), T_{add_bool_wdgt}(2), \dots T_{add_bool_wdgt}(N), N=16$
- $T_{rmv_bool_wdgt}(1), T_{rmv_bool_wdgt}(2), \dots T_{rmv_bool_wdgt}(N), N=16$
- $T_{rmp_bool_wdgt}(1), T_{rmp_bool_wdgt}(2), \dots T_{rmp_bool_wdgt}(N), N=16$
- $T_{add_str_wdgt}(1), T_{add_str_wdgt}(2), \dots T_{add_str_wdgt}(N), N=16$
- $T_{rmv_str_wdgt}(1), T_{rmv_str_wdgt}(2), \dots T_{rmv_str_wdgt}(N), N=16$
- $T_{rmp_str_wdgt}(1), T_{rmp_str_wdgt}(2), \dots T_{rmp_str_wdgt}(N), N=16$
- T_{gen_frm}
- $T_{gen_btn}(1), T_{gen_btn}(2), \dots T_{gen_btn}(N), N=42$
- $T_{gen_num_wdgt}(1), T_{gen_num_wdgt}(2), \dots T_{gen_num_wdgt}(N), N=16$
- $T_{gen_bool_wdgt}(1), T_{gen_bool_wdgt}(2), \dots T_{gen_bool_wdgt}(N), N=16$
- $T_{gen_str_wdgt}(1), T_{gen_str_wdgt}(2), \dots T_{gen_str_wdgt}(N), N=16$

For each sub-operation, N represents an integer that is at least the maximum input value to which the generator will be exposed during interaction. We looked at all cases of retargeting and generating user-interfaces of our six networked devices to pick the N values.

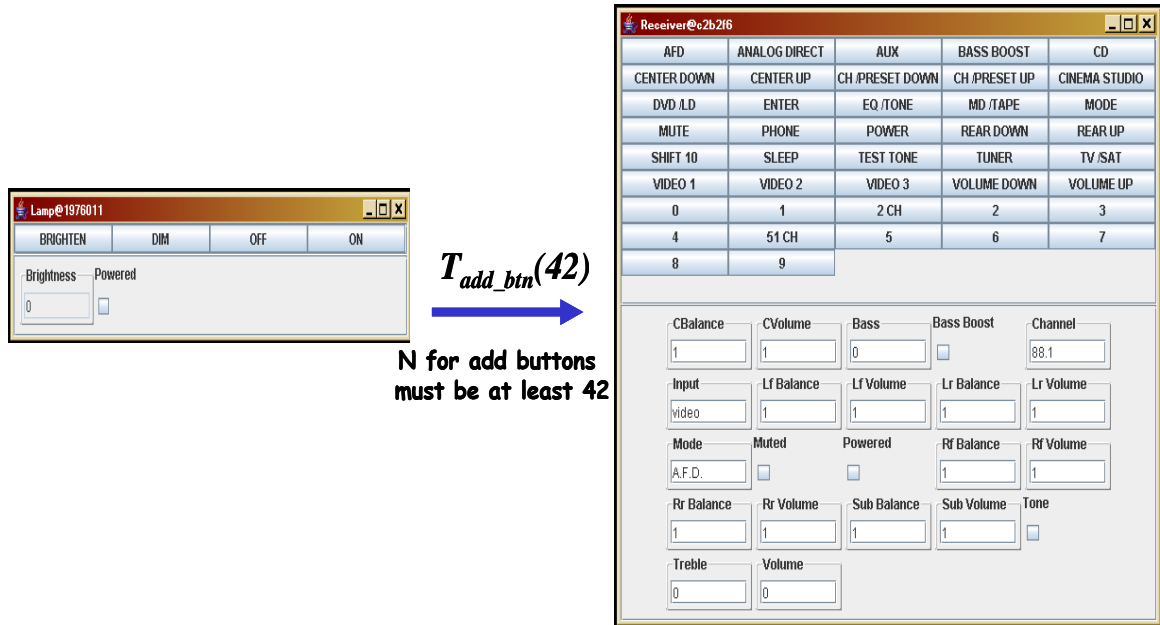


Figure 42. The maximum N value for T_{add_btn} comes from retargeting a light UI to a receiver.

To illustrate this profiling process, consider the task of gathering the data for T_{add_btn} from one to N . The maximum N value is forty-two, which is the count for the number of buttons to add when retargeting the lamp user-interface to the receiver (Figure 42). This process represents the case in which the most buttons are added to a source user-interface to fit a target device. Thus, our measurements included the individual times for adding a set of one to forty-two buttons to an empty GUI (Figure 43a). Adding, remapping, and removing zero widgets inherently has no time cost. Similarly, the maximum N value for T_{rmv_btn} is forty-two, which is the count for the number of buttons to remove when retargeting the receiver user-interface to the lamp. This process represents the case in which the most buttons are removed from a source user-interface to fit a target device.

Therefore, we collected the times for removing a set of one to forty-two buttons from a user-interface (Figure 43b). T_{rmv_btn} 's maximum N value is also forty-two, which is the number of buttons to remap when retargeting a receiver's user-interface to a another receiver with an identical set of commands. This process represents the case in which the most buttons on a source user-interface are remapped to target device. Hence, we gathered the times for retargeting a set of 1 to forty-two pre-existing buttons on a user-interface (Figure 43c). To measure the times for adding, removing, and remapping sets of property widgets, we followed a similar process as just described for buttons. It is important to mention that for each profiling experiment, we chose its respective N value to be at or just above the maximum value possible given the six devices. To generically support devices, a very large N value would be chosen.

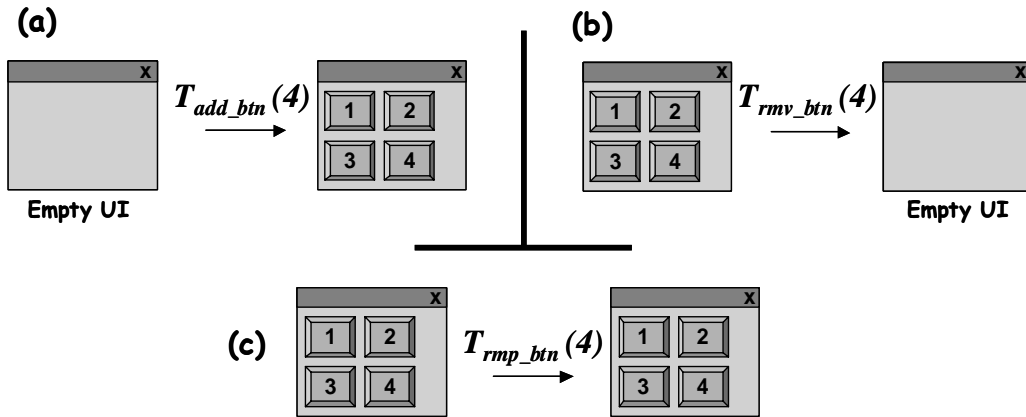


Figure 43. A depiction of part of our profiling experiments: (a) finding the time it takes to add 4 buttons to an empty UI, (b) finding the time it takes to remove 4 buttons from a UI, and (c) finding the time it takes to remap 4 buttons on a UI.

We performed all the profiling experiments mentioned above on the same laptop that we used to gather most of the deployment time data presented in the previous chapter {733 MHz Pentium – 128MB}. As our experiments in the previous chapter show, a client's computation power directly affects its generation time. To evaluate how much a client's computation power can affect its profiling times, we also collected some data on a slower laptop {400Mhz Celeron - 64MB} (Figures 44-47). The fast laptop yields times that are significantly lower than the slow laptop. For instance, the slow laptop

takes approximately seven times longer than the fast laptop to add forty-two buttons to an empty user-interface.

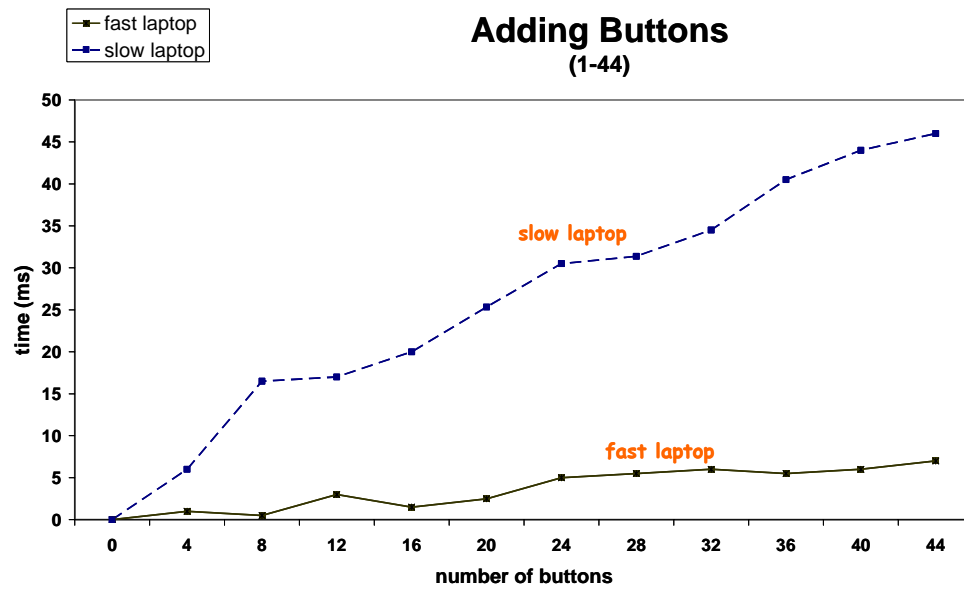


Figure 44. The time it takes to add new buttons to an empty GUI (slow vs. fast laptop).

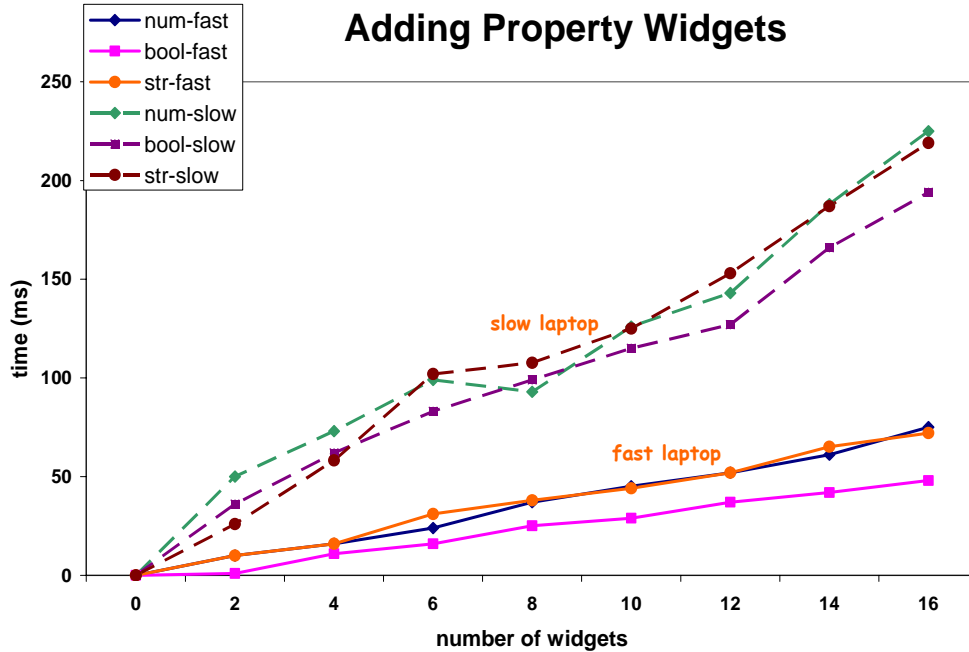


Figure 45. The time it takes to add new property widgets to an empty GUI (slow vs. fast laptop). The dashed lines correspond to the slow laptop and the bold lines correspond to the fast laptop.



Figure 46. The time it takes to remove pre-existing property widgets on a GUI (slow vs. fast laptop)

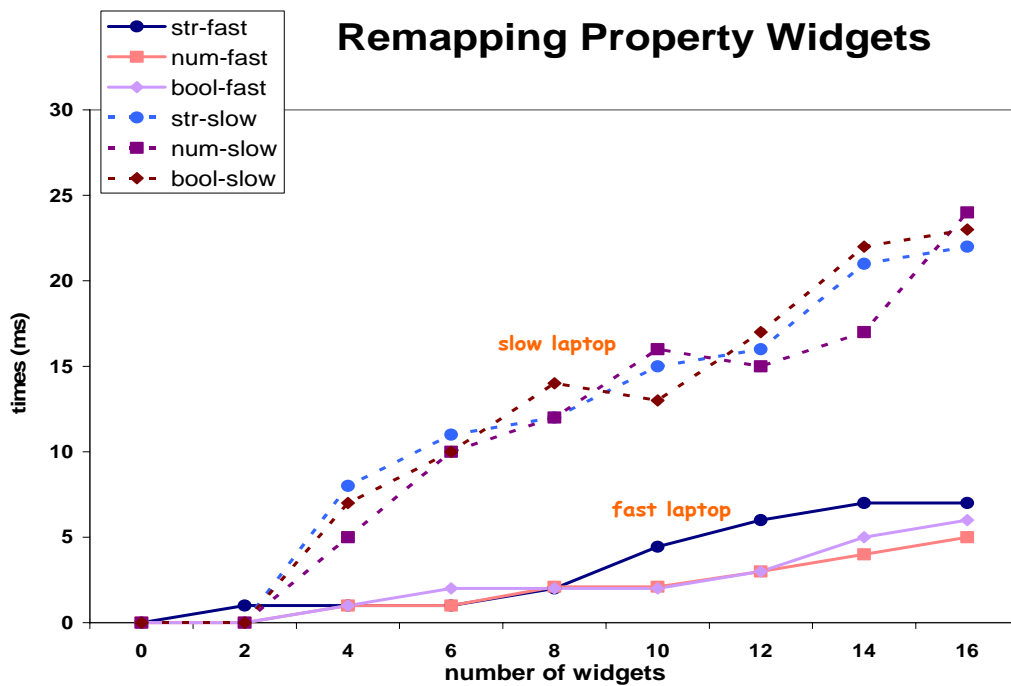


Figure 47. The time it takes to remap pre-existing property widgets on a UI (slow vs. fast laptop)

We also collected some times on same Ipaq mentioned in the previous chapter. They were much longer than their corresponding fast and slow laptop times. For example, it can take nearly five times longer to remap a property widget on the Ipaq than on the slow laptop (Figure 48). Compared to the fast laptop, the Ipaq can take over twenty times longer than the fast laptop. The time differences between the three clients imply that generation and retargeting estimation functions must be device specific.

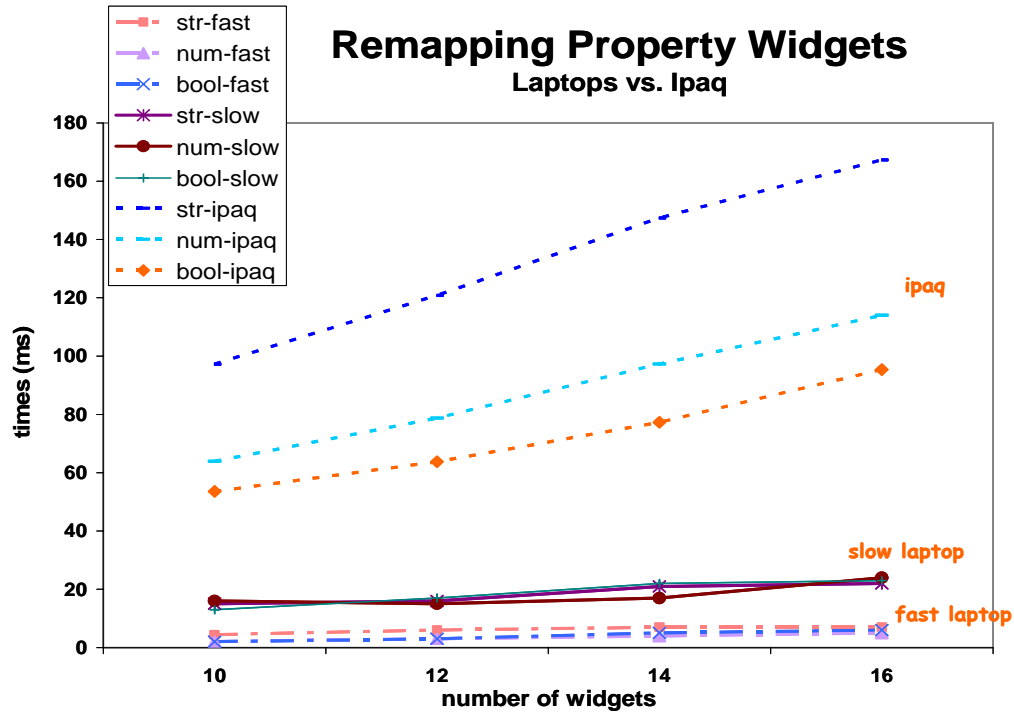


Figure 48. A graph illustrating time differences between the three clients. It also shows that widgets representing different types can yield different operation times—particularly on the Ipaq.

Yet another important conclusion of the data is the weight the retargeting sub-operations have on execution time. On the fast laptop, the button removal and remapping operations have no time costs up to their respective N values. Adding new buttons, however, has a significant time cost. Also, it takes more time to add a new property widget to a user-interface than remap an already existing one of the same kind (Figure 49). These results support our general argument of saving time by retargeting parts of a user-interface instead of generating new ones.

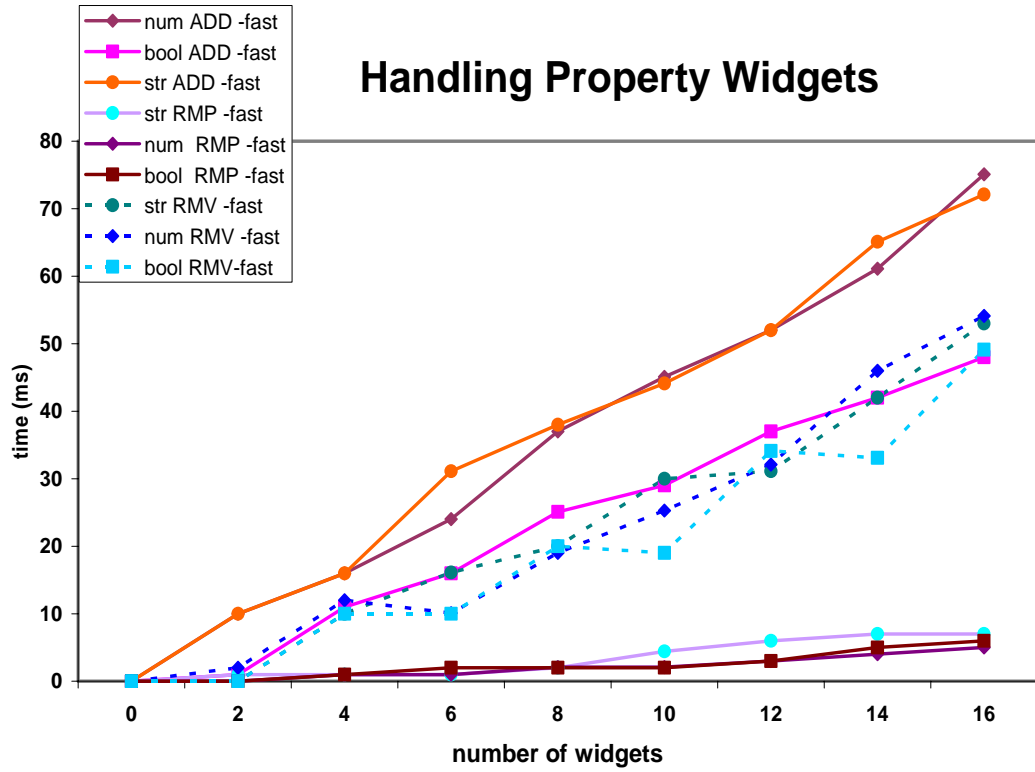


Figure 49. The differences between adding, removing, and remapping a property widget.

Using the collected data, we performed the actual regression operations to derive the estimation functions for the fast laptop. We applied MATLAB's polynomial fitting command (called `polyfit`) on each sub-operation's data set to find its empirical time-cost function. This command takes in a domain and range of data and returns a polynomial function of a given degree that estimates the data. In our case, the domain of each sub-operation's function is the set of workload values we used to get actual execution times. The range is the corresponding set of execution times. Given the linear behavior of the data, we chose a degree of one for each function. Below are the two functions (T_{ret} and T_{gen}) for the fast laptop that we derived using the sub-operation functions returned from `polyfit` and simplifying:

- 1) $T_{ret}(BA, NWA, BWA, SWA, NWD, BWD, SWD, NWR, BWR, SWR) =$
 $0.16BA + 6.17NWA + 3.63BWA + 5.67SWA + 3.37NWD + 2.94BWD +$
 $3.28SWD + 0.31NWR + 0.36BWR + 0.50SWR - 35.22$
- 2) $T_{gen}(BG, NWG, BWG, SWG) = 2.70BG + 6.17NWG + 3.63BWG + 5.67SWG + 22.24$

The entire process that led to the two functions could be automated by the notion of a *self-profiling generator*. Such a generator would run a bootstrap program that automatically performs all the necessary profiling operations and measures its own performance on a given client. The program would then run regression code, as implemented by MATLAB's `polyfit` operation, to return a T_{ret} and T_{gen} function specifically for the client.

As implied by the ten parameters required by the above T_{ret} function, a problem of *regression-based source-device prediction* is long search times. In order to predict the fastest (source) user-interface to retarget, a generator must search through the commands and properties of the source and target devices to determine parameter values for T_{ret} . If the target and source devices are complex or there are many potential source user-interfaces available, searching can become an expensive process. For this reason, we support the idea of cache-based retargeting.

A generator caches the T_{ret} value it calculates for each source and target device type pair it ever evaluates. If a pair with a cached T_{ret} value occurs again in the system, the generator avoids recalculating $T_{ret}()$, which involves searching the programming interfaces of its respective types to find values the functions ten parameters. Instead, it simply retrieves the stored value. Notice the similarity between the parameter values required by $T_{ret}()$ and the retarget lists *TOC*, *SOC*, *SC*, *TOP*, *SOP*, and *SP* for `retarget()`. T_{ret} requires the amount of buttons and state widgets that must be added, removed, and remapped to change a source user-interface to fit the target device. The `retarget()` method requires lists are the names of commands and properties that correlate to these same buttons and state widgets that must be added, removed, and remapped. In determining T_{ret} 's parameter values, the algorithm inherently builds a list containing these names. That is, the lists are a byproduct of gathering $T_{ret}()$'s parameters. These lists are thus cached so that they can be accessed and passed to `retarget()` if the generator decides to retarget.

On occasions where there are multiple potential source user-interfaces to retarget to a target device, the generator also caches the corresponding source device type of user-

interface that it predicts to be the fastest. If a user wants to use a device of the target's type later and the same set of source user-interfaces is available, the generator directly picks the user-interface associated with the cached device type. Thus, it avoids all the operations involved in finding the fastest user-interface. The generator also caches T_{gen} values for each target device type to avoid repeating the process of gathering the function's parameters.

Given this overview of cache-based retargeting, we can now show the algorithm for the method `retargetORgenerate()` which uses possibly cached information to: (1) select the fastest source user-interface to retarget, (2) decide whether to retarget or generate, and (3) retrieve retargeting lists TOC , SOC , SC , TOP , SOP , and SP needed by `retarget()`, defined earlier, if it predicts generation to be slower. In presenting `retargetORgenerate()` we also show two 'helper methods' it invokes if retargeting is found optimal—`retargetHomogeneous()` and `retargetHeterogeneous()`.

Let $C1$ = a cache. For a given set of source device types (S) and target device type (t), $C1$ caches: (1) the decided source device type ($s_{fastest}$) in S with the lowest T_{ret} value (T_{ret}^{min}) and (2) the actual T_{ret}^{min} value. Imagine $C1$ as a hash table: $C1.put([s, t], [s_{fastest}, T_{ret}^{min}])$ inserts the elements in the cache and $C1.get([s, t])$ returns the cached collection $[s_{fastest}, T_{ret}^{min}]$

Let $C2$ = a cache. For a given source device type (s) and target device type (t), $C2$ stores the results from calculating T_{ret} and 'retargeting lists' $\{TOC, SOC, SC, TOP, SOP, SP\}$. Imagine $C2$ as a hash table: $C2.put([s, t], [T_{ret}, TOC, SOC, SC, TOP, SOP, SP])$ inserts the elements in the cache and $C2.get([s, t])$ returns the cached collection $[T_{ret}, TOC, SOC, SC, TOP, SOP, SP]$

Let $C3$ = a cache. For a given target device type (t), $C3$ stores the type's T_{gen} value. Imagine $C3$ as a hash table: $C3.put(t, T_{gen})$ inserts the elements in the cache and $C3.get(t)$ returns T_{gen} .

Let S = the set of device types of the currently available source UIs

Let U = the set of source user-interface objects

Let $target$ = the target device

```
boolean retargetHomogeneous(U,t) {
    match = getMatchingSourceUIforType(U,t);
    if (match != null) {
        retarget(target, match, null, null, getCommandNames(t), null, null,
```

```

        getPropertyNames( $t$ )
    return true
}
else
    return false
}

boolean retargetHeterogeneous( $C1, C2, C3, S, U, t$ ) {
    [ $s_{fastest}, T_{ret}^{min}$ ] =  $C1.get([S, t])$ 
    if ([ $s_{fastest}, T_{ret}^{min}$ ] == null) {
        for each source device type ( $s$ ) in  $S$  {
            [ $T_{ret}, TOC, SOC, SC, TOP, SOP, SP$ ] =  $C2.get([s, t])$ 
            if ([ $T_{ret}, TOC, SOC, SC, TOP, SOP, SP$ ] == null) {
                [ $T_{ret}, TOC, SOC, SC, TOP, SOP, SP$ ] = computeTret( $s, t$ )
                 $C2.put([s, t], [T_{ret}, TOC, SOC, SC, TOP, SOP, SP])$ 
            }
            [ $s_{fastest}, T_{ret}^{min}$ ] = min ( $s_{fastest}, T_{ret}^{min}, s, T_{ret}$ )
        }
         $C1.put([S, t], [s_{fastest}, T_{ret}^{min}])$ 
    }
     $T_{gen} = C3.get(t)$ 
    if ( $T_{gen} == null$ ) {
         $T_{gen} = computeTgen(t)$ 
         $C3.put(t, T_{gen})$ 
    }
    if ( $T_{ret}^{min} \leq T_{gen}$ ) {
        [ $TOC, SOC, SC, TOP, SOP, SP$ ] = extractRetargetingLists( $C2.get([s_{fastest}, t])$ )
        retarget( $target$ , getMatchingSourceUIforType( $U, s_{fastest}$ )  $TOC, SOC, SC, TOP, SOP, SP$ )
        return true
    }
    else
        return false
}

```

```

retargetORgenerate( $C1, C2, C3, S, U, target$ ) {
     $t = getType(target)$ 
    if (retargetHomogeneous( $U, t$ ))
        return
    else {
        if retargetHeterogeneous( $C1, C2, C3, S, U, t$ )
            return
        else
            generateUI( $target$ )
    }
}

```

The `retargetORgenerate()` accepts the references to: cache $C1$, cache $C2$, the set of source device types (S), the set of source user-interface objects (U), and the target device ($target$). It assumes that the fastest user-interface to retarget is always the one created from a source device that is the same type as the target device (t). Thus, it first checks

for such a user-interface by first calling `retargetHomogeneous()`. Further, this method assumes that retargeting this user-interface is always faster than generating a new one because retargeting would only involve remapping user-interface components.

Given t , `retargetHomogeneous()` calls `getMatchingSourceUIforType(U, t)`, which searches the set of source user-interface objects to see whether one has already been created for a source device of type t . If such a user-interface object exists, `retargetHomogeneous()` calls `retarget()` to actually retarget the object. It then returns true, notifying `retargetORgenerate()` that it performed the retargeting. Notice the null values passed into `retarget()` for TOC , SOC , TOP , and SOP . The reason for them is that when retargeting a user-interface between two devices of the same type there are no buttons and property widgets to add and remove. All components are shared.

If `getMatchingSourceUIforType(U, t)` does not return a matching user-interface, then `retargetHomogeneous()` returns false. The result of `retargetHomogeneous()` decides the next step in `retargetORgenerate()`. With a true result, `retargetORgenerate()` terminates since `retargetHomogeneous()` completed the actual retargeting. Otherwise, it must decide whether to: (1) retarget the fastest source user-interface created for a device of a different type than the target or (2) generate a new one.

To decide on which approach to take, `retargetORgenerate()` calls `retargetHeterogeneous()`, which accepts CI , $C2$, S , U , and t . The first step of `retargetHeterogeneous()` is to check the cache CI to see if the specific set S and type t have been previously evaluated to find the source device type ($s_{fastest}$) that yields the lowest T_{ret} value (T_{ret}^{min}). If so, it stores the $s_{fastest}$ and T_{ret}^{min} value from the cache in a variable. Otherwise, the method begins searching for $s_{fastest}$ and T_{ret}^{min} . This involves getting the T_{ret} value for each source and target device type pair (s, t) produced by S and t . The pair with the lowest T_{ret} value (T_{ret}^{min}) contains $s_{fastest}$. It finds these values by first checking the cache $C2$ to see if a given pair has been previously evaluated to find its T_{ret} value and the corresponding retargeting lists TOC , SOC , SC , TOP , SOP , and SP . If so, then it stores this collection in a variable. Otherwise, it must call `computeTret()` to determine these values. Given s and t this method searches their programming interfaces

to determine the parameter values needed for $T_{ret}()$ and then calculates the function's value. Recall that the retargeting lists for the pair is a byproduct of this process and is thus returned with T_{ret} . Thus, `computeTret()` returns a collection consisting of T_{ret} and the retargeting lists. This returned collection is inserted into cache *C2*. As `retargetHeterogeneous()` evaluates each pair (s,t) , it uses `min()` to keep track of the $s_{fastest}$ it has seen so far with the lowest T_{ret} value (T_{ret}^{min}). After it is done evaluating each pair (i.e. the loop), the final $s_{fastest}$ and T_{ret}^{min} are appropriately defined for (S,t) . It inserts this information in the cache *C1*.

With $s_{fastest}$ and the corresponding T_{ret} value decided for (S,t) , `retargetHeterogeneous()` moves on to decide whether to retarget or generate. It checks cache *C3* to see whether T_{gen} for the target device's type (t) has ever been calculated. If so, then it stores this value. Otherwise, it must search the programming interface of type t to get $T_{gen}()$'s needed parameter values. It then calculates the function's value. At this point, the method knows $s_{fastest}$, T_{ret} , and T_{gen} . If T_{ret} is lower or equal to T_{gen} , it executes `retarget()`, passing in the reference to the target device, source user-interface generated for $s_{fastest}$, and retargeting lists. It then returns true, notifying `retargetORgenerate()` that it performed the retargeting. Otherwise, it generates a new user-interface from the target device's reference.

4.3 SUI Retargeting

Our SUI generator retargeting implementation is much more basic than `ObjectEditor`'s. Like Hodes' GUI generator, it can only retarget when there is a source device that is of the same type as the target device. The algorithm it follows is simple:

Let target = the target device

Let current = the source device type

```

retarget(current, target) {
    if (current == getType(target)) {
        RECOGNIZER.suspend();
        setRPCreference(target)
        RECOGNIZER.resume();
        SYNTHESIZER.speakText("start talking");
    }
}

```

The generator can only support one SUI at a time. Thus, there is always just one potential source user-interface. The method `retarget()` accepts the source device's type (*current*) and a reference to the target device (*target*). It checks to see if the target device and source device are of the same type. If so, it suspends the recognizer for a moment so that it can switch the RPC reference of the user-interface to the target device. It then resumes the recognizer and notifies the user to 'start talking'.

4.4 Evaluation

We evaluated our retargeting approach using the ObjectEditor and SUI generator implementations described above. This evaluation focuses on three important performance criteria:

- 1) *Source User-Interface Selection Performance* - In scenarios where multiple source user-interfaces are available for retargeting, is the one with the lowest T_{ret} value actually the fastest?
- 2) *Approach Selection Performance* - Does picking the lower value between T_{ret} and T_{gen} accurately decide the faster approach—retarget or generate?
- 3) *Retargeting Performance* - Can retargeting actually offer deployment times that are comparable to the client-factory approach?

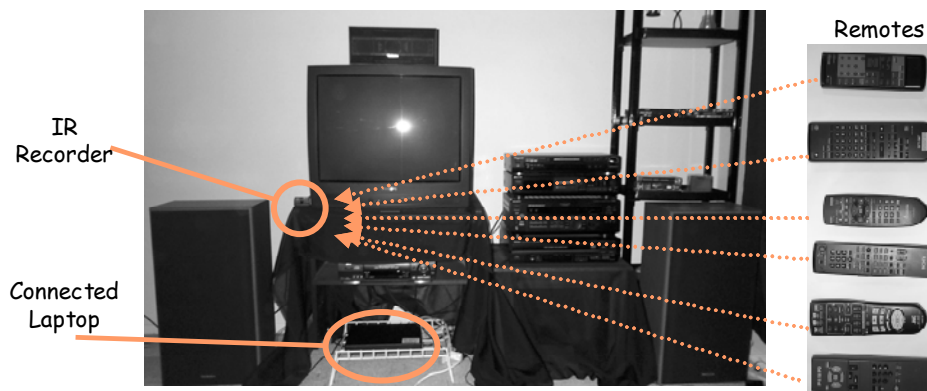


Figure 50. Our tool for automatically recording device interactions at a person's home.

To answer these questions, we wanted to use real world sequences of device accesses. Thus, we collected interaction data from different users performing their device-related tasks. We gathered the data in two kinds of environments where people frequently use devices—at their respective homes and a conference room in our building. To gather such data, we built a tool that records the IR-based interactions offered by traditional remote controls. This tool consists of an *Evation IRMan* device connected to a laptop’s serial port (Figure 50 and 51). An *IRMan* captures an infrared signal from a remote control and outputs an ASCII string code representing the signal. A programming running on the laptop reads the code and maps it to a string representing the associated device and command (e.g. ‘VCR.play’). It then stores the string, along with the time and date of the invocation, on the laptop’s disk. This tool avoids relying on people to self-monitor themselves, which is cumbersome and can introduce human-error. Also, it is unobtrusive and mobile, which allows for easy setup. Its limitation is that it only records interactions within a single room at a time because IR signals cannot pass through walls.



Figure 51. A close up of the IRman serial port device.

Given this limitation, we used the tool to record interactions within the entertainment centers of our participants. Entertainment centers typically contain several devices that people often use at home. Table 6 describes the users we recruited and the specific devices they own. We logged each of them for a period varying from one to two weeks, producing a total of well over 30,000 recorded commands. As other chapters of this dissertation will show, this data is valuable beyond its use in evaluating our retargeting approach.

User	Summary			
	Gender	IR Devices	Age	Education/Employment
P1	Male	TV, VCR, DVD player	25	Final semester masters student in computer science. Full-time computer programmer.
P2	Male	TV, DVD changer, Receiver	30	High-school graduate. Food server in a restaurant.
P3	Female	TV/VCR combo, cable box	23	Second year Ph.D student in sociology.
P4	Male	TV, DVD player	25	Second year medical student.
P5	Female	TV, DVD player, stereo system	23	College graduate in journalism. Works full-time in advertising.
P6	Female	TV, VCR/DVD combo	27	Second year Ph.D student in biostatistics.
P7	Male	TV, VCR, DVD player, Receiver, XBOX	27	Masters degree in computer science. Full-time programmer.
P8	Female	TV/VCR combo, DVD player	26	Second year law student.
P9	Male	TV,DVD player, stereo system	24	College graduate in marketing. Full-time mortgage analyst.
P10	Male	TV, cable box w/ built-in TIVO	24	College graduate in marketing. Unemployed.
P11 (the author)	Male	TV, VCR, DVD player, Receiver	26	Senior Ph.D student in computer science.

Table 6. A summary of our 11 participants.

The conference room, on the other hand, is a static environment and consists of consisted of a projector and three lamp arrays (Figure 52). Each lamp array is basically a set of two or more individual lamps that are controlled by one unifying switch. We were especially interested in the task of ‘setting up for a presentation’, which involves a series of deterministic device accesses. Thus, we did not need to use our IR recording mechanism in this room. Essentially, the task involves turning on the lamps in the room, setting up the projector, and then dimming (or turning off) the lamps.

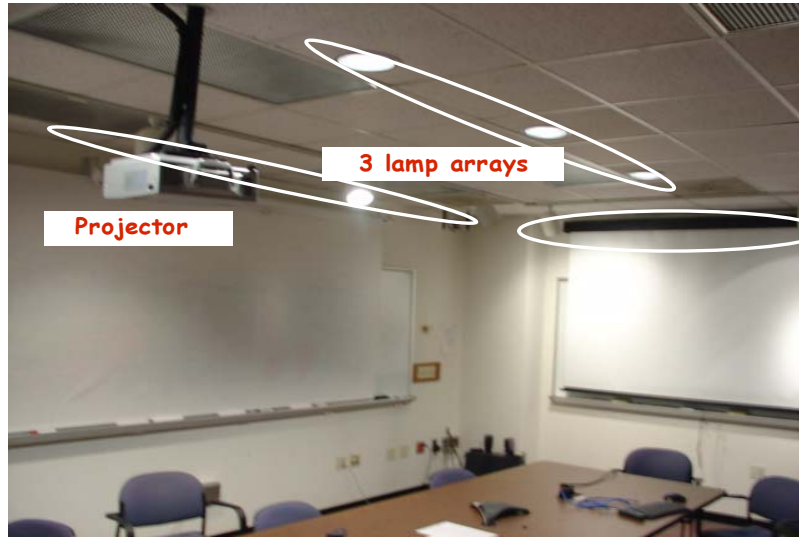


Figure 52. The conference room we used.

4.4.1 Source User-interface Selection Performance

In scenarios where multiple source user-interfaces are available for retargeting, is the one with the lowest T_{ret} value actually the fastest to retarget?

Our first step in evaluating T_{ret} 's prediction performance was to identify a benchmark set of scenarios. We were able to use our participants' logs to produce this set. From these logs, we could determine when a participant goes from one task directly to another. Assuming the participants had client computers that could perform retargeting, the user-interfaces from the previous task could immediately be available as sources for the next. To illustrate, imagine a person who watches cable TV for a while and then watches a DVD. The TV and cable box user-interfaces would serve as possible sources to retarget to the DVD player; i.e., the mapping: (TV UI, cable box UI \rightarrow DVD player). A question that arises from this example is: how does the client have both the TV and cable box user-interface available and not just the (retargeted) user-interface of the last accessed device? Ideally, during a task, all device user-interfaces that are associated with the task should be active in memory so that a user can directly switch back and forth between them. We imagine a generator accepting a list of devices involved in a user's desired task. For each device in the list, the generator would create a new user-interface if all existing user-interfaces are already associated with other devices on the list.

In order to discover our participants' specific task transitions, we interviewed them and examined their logs for the set of tasks they performed and the device-commands invoked in each task. Using this information, we searched the logs for task transitions. We only considered the logs of users who owned the types of IR devices that we networked (TV, lamps, DVD player, VCR, and A/V receiver). The reason is that evaluating T_{ret} 's performance requires actual networked devices, and there would be too much overhead in individually networking the devices of all of our participants. Thus, we had to use our networked devices to simulate those of the participants with matching types.

We found three unique transitions within the logs, with many of the participants producing the same cases. Using the presentation room example, we supplemented these examples with two more. In particular, we imagined a user who enters a conference room and has the user-interfaces of four commonly used home devices (a TV, VCR, Receiver, and lamp) still running on a client. The four user-interfaces are thus available for retargeting to the projector and lamp arrays in the room. One can imagine a user having these user-interfaces loaded because he or she just came from home or was monitoring how children at home use the devices.

For each of the five total transitions, Table 7 compares the command-only user-interface with the lowest T_{ret} value to the command-only user-interface we actually measured to be the fastest to retarget. Table 8 makes a similar comparison for deploying command-and-state based user-interfaces. The two tables show that prediction using T_{ret} correctly picks the fastest user-interface for all five cases regardless of the kind of user-interface being deployed.

For each task transition, Tables 7 and 8 also show the difference (actual and percentage) between the retargeting time of the actual fastest source user-interface and each of the other available ones. These differences represent the benefits in selecting the actual fastest source user-interface. To illustrate, for the 'turning on conference room lights' task transition of Table 7, there is a 95% (or 116.73 ms) increase in retargeting time when choosing the receiver's user-interface over the lamp's. Given that such large

differences can occur, it is important to select the source user-interface that is actually the fastest. The small differences, on the other hand, show the precision of *regression-based source-device prediction*. As Tables 7 and 8 respectively show, this approach is successful even when there is only a two and one percent difference between the retargeting times of two potential source user-interfaces.

Task Transition	Mapping (source UIs → target device)	Prediction (lowest T_{ret})	Actual (fastest measured)	Percentage Difference (Slow-Fast)/Slow	Actual Difference(ms) Fast-Slow
Watch TV after DVD movie	TV,DVD,RCVR → VCR	DVD UI→VCR	DVD UI→VCR	(TV -DVD)=19%, (RCVR - DVD)=23%	TV-DVD= 18.50; RCVR-DVD= 23.22
Watch DVD movie or listen to music after watching TV*	TV, RCVR → DVD	TV UI→DVD	TV UI→DVD	(RCVR - TV)=24%	RCVR-TV=19.72
Watch DVD movie or listen to music after watching TV*	TV,VCR,RCVR → DVD	VCR UI→DVD	VCR UI→DVD	(RCVR - VCR)=40%, (TV - VCR)=20%	RCVR-VCR=32.50, TV-VCR= 12.78
Turn lights on in presentation room	TV,VCR,RCVR,LAMP→ LAMP	LAMP UI→ LAMP	LAMP UI→ LAMP	(RCVR - LAMP)=95%, (VCVR - LAMP)=88%, (TV - LAMP)=91%	RCVR-LAMP=116.73, VCR-LAMP=21.79, TV-LAMP=57.73
Setup projector in presentation room	TV,VCR,RCVR,LAMP→ PROJ	TV UI→PROJ	TV UI→PROJ	(LAMP -TV)=39%, (VCR - TV)=2%, (RCVR - TV)=35%	LAMP-TV= 32.34, VCR-TV= 1.28, RCVR-TV= 27.53

Table 7. An evaluation of Tret’s ability to predict the fastest command-only user-interface to retarget. { * The DVD player also serves as a music CD player }

Task Transition	Mapping (source UIs → target device)	Prediction (lowest T_{ret})	Actual (fastest measured)	Percentage Difference (Slow-Fast)/Slow	Actual Difference(ms) Fast-Slow
Watch TV after DVD movie	TV,DVD,RCVR → VCR	TV UI→VCR	TV UI→VCR	(DVD -TV)=24%, (RCVR - TV)=42%	DVD-TV=48.78; RCVR-TV= 105.78
Watch DVD movie or listen to music after watching TV*	TV, RCVR → DVD	TV UI→DVD	TV UI→DVD	(RCVR - TV)=13%	RCVR-TV=38.84
Watch DVD movie or listen to music after watching TV*	TV,VCR,RCVR → DVD	VCR UI→DVD	VCR UI→DVD	(RCVR - VCR)=15%, (TV - VCR)=1%	RCVR-VCR=42.49, TV-VCR=3.75
Turn lights on in presentation room	TV,VCR,RCVR,LAMP→ LAMP	LAMP UI→ LAMP	LAMP UI→ LAMP	(RCVR - LAMP)=93%, (VCVR - LAMP)=83%, (TV - LAMP)=86%	RCVR-LAMP=129.89, VCR-LAMP=76.81, TV-LAMP=69.11
Setup projector in presentation room	TV,VCR,RCVR,LAMP→ PROJ	LAMP UI→PROJ	LAMP UI→PROJ	(TV - LAMP)=10%, (VCR - LAMP)=18%, (RCVR - LAMP)=31%	TV-LAMP=12.23, VCR-LAMP = 24.45, RCVR-LAMP=49.89

Table 8. An evaluation of Tret’s ability to predict the fastest command-and-state based user-interface to retarget. { * The DVD player also serves as a music CD player }

4.4.2 Approach Selection Performance

Does picking the lower value between T_{ret} and T_{gen} accurately decide the faster approach—retarget or generate?

Tables 9 and 10 respectively evaluate this selection method for command-only and command-and-state based user-interface deployment. For each of the five identified transitions, the tables show the approach predicted to be the fastest and the approach that is actually the fastest. The results show that selection using the lower value of T_{ret} and T_{gen} correctly picks the fastest approach for all cases. In fact, as the percentage differences on the two tables show, retargeting is always at least twice as fast.

Task Transition	Mapping (source UIs \rightarrow target device)	Prediction	Actual	Percentage Difference (Gen-Ret)/Gen	Actual Difference(ms) Gen-Ret
Watch TV after DVD movie	TV,DVD,RCVR \rightarrow VCR	Retarget: DVD UI \rightarrow VCR	Retarget: DVD UI \rightarrow VCR	79%	296.71
Watch DVD movie or listen to music after watching TV*	TV, RCVR \rightarrow DVD	Retarget: TV UI \rightarrow DVD	Retarget: TV UI \rightarrow DVD	87%	410.22
Watch DVD movie or listen to music after watching TV*	TV,VCR,RCVR \rightarrow DVD	Retarget: VCR UI \rightarrow DVD	Retarget: VCR UI \rightarrow DVD	90%	423.00
Turn lights on in a presentation room	TV,VCR,RCVR,LAMP \rightarrow LAMP	Retarget: LAMP UI \rightarrow LAMP	Retarget: LAMP UI \rightarrow LAMP	98%	306.69
Setup projector in presentation room	TV,VCR,RCVR,LAMP \rightarrow PROJ	Retarget: TV UI \rightarrow PROJ	Retarget: TV UI \rightarrow PROJ	86%	308.00

Table 9. For command-only UI deployment, a comparison of the approach predicted to be the fastest to the approach that actually measures to be the fastest.

Task Transition	Mapping (source UIs \rightarrow target device)	Prediction	Actual	Percentage Difference (Gen-Ret)/Gen	Actual Difference (ms) Gen-Ret
Watch TV after DVD movie	TV,DVD,RCVR \rightarrow VCR	Retarget: TV UI \rightarrow VCR	Retarget: TV UI \rightarrow VCR	72%	397.47
Watch DVD movie or listen to music after watching TV*	TV, RCVR \rightarrow DVD	Retarget: TV UI \rightarrow DVD	Retarget: TV UI \rightarrow DVD	64%	451.95
Watch DVD movie or listen to music after watching TV*	TV,VCR,RCVR \rightarrow DVD	Retarget: VCR UI \rightarrow DVD	Retarget: VCR UI \rightarrow DVD	65%	455.7
Turn lights on in a presentation room	TV,VCR,RCVR,LAMP \rightarrow LAMP	Retarget: LAMP UI \rightarrow LAMP	Retarget: LAMP UI \rightarrow LAMP	97%	330.45
Setup projector in presentation room	TV,VCR,RCVR,LAMP \rightarrow PROJ	Retarget: LAMP UI \rightarrow PROJ	Retarget: LAMP UI \rightarrow PROJ	78%	388

Table 10. For command-and-state based UI deployment, a comparison of the approach predicted to be the fastest to the approach that actually measures to be the fastest.

4.4.3 Retargeting Performance

Can retargeting actually offer times that are at least as low as client-factory times?

Using our real world scenarios, we evaluate how close two important levels of retargeting can achieve client-factory-like times—homogeneous and heterogeneous retargeting. In homogeneous retargeting we assume that a source device that is the same type as the target device is always available. In other words, it expects that (in memory) there is a source user-interface object previously made for a device that is of the target device’s exact type. Heterogeneous retargeting avoids this assumption by supporting different types (or programming interfaces).

We also consider the effects of different levels of client processing power on retargeting time. The differences between the Ipaq and fast laptop deployment times for all approaches (client-factory, remote-factory, and generation), presented in the previous chapter, motivate this analysis. We use the same Ipaq and laptop. Using these same devices also allows us to appropriately compare their retargeting times with their client-factory times.

Another factor that we found to effect deployment time is the kind of user-interface being deployed. As a result, we consider retargeting times for: (a) command-only user-interfaces, (b) command-and-state based user-interfaces, (c) SUIs, and (d) GUIs (included in (a) and (b)). Yet another factor shown to effect deployment time is the network speed available to the client. Like in previous experiments, we compare retargeting times using a wired LAN and dialup connection.

4.4.3.1 Homogeneous Retargeting

Homogeneous retargeting implies low deployment times. As described earlier, retargeting a user-interface built for the target device’s type only requires remapping its components. No new components need to be created. Figures 52-54 respectively compare the homogeneous retargeting times for our device’s command-only GUIs, command-and-state based GUI, and command-only SUIs to the times of alternate approaches. We collected all the times on the laptop using a wired LAN connection.

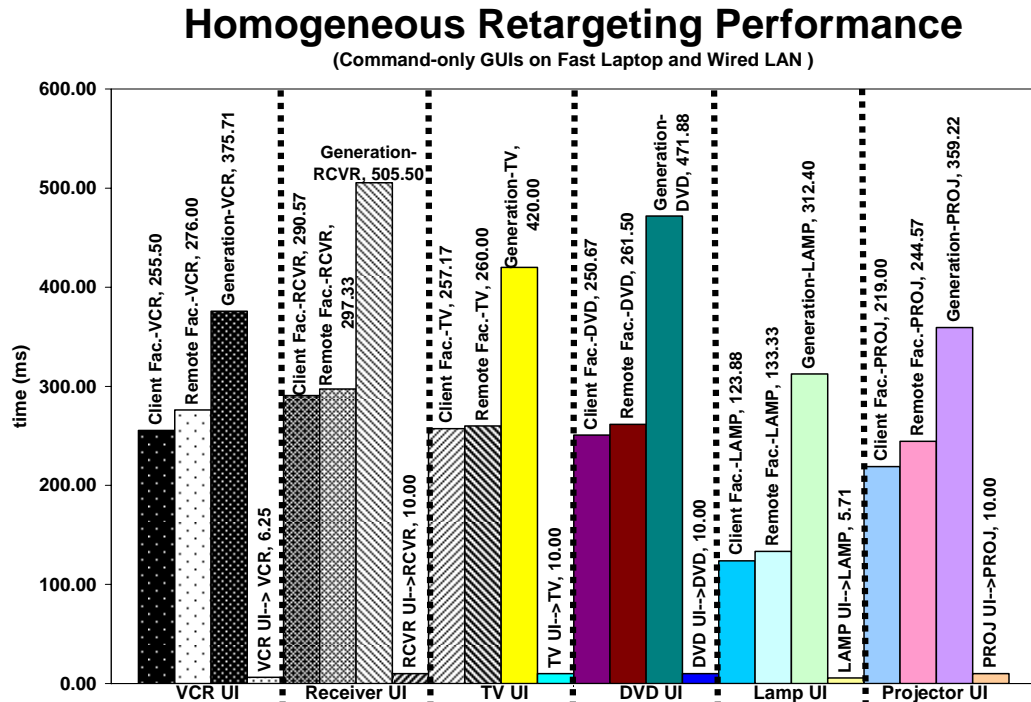


Figure 52. Homogeneous retargeting of command-only GUIs using the fast laptop and wired LAN connection.

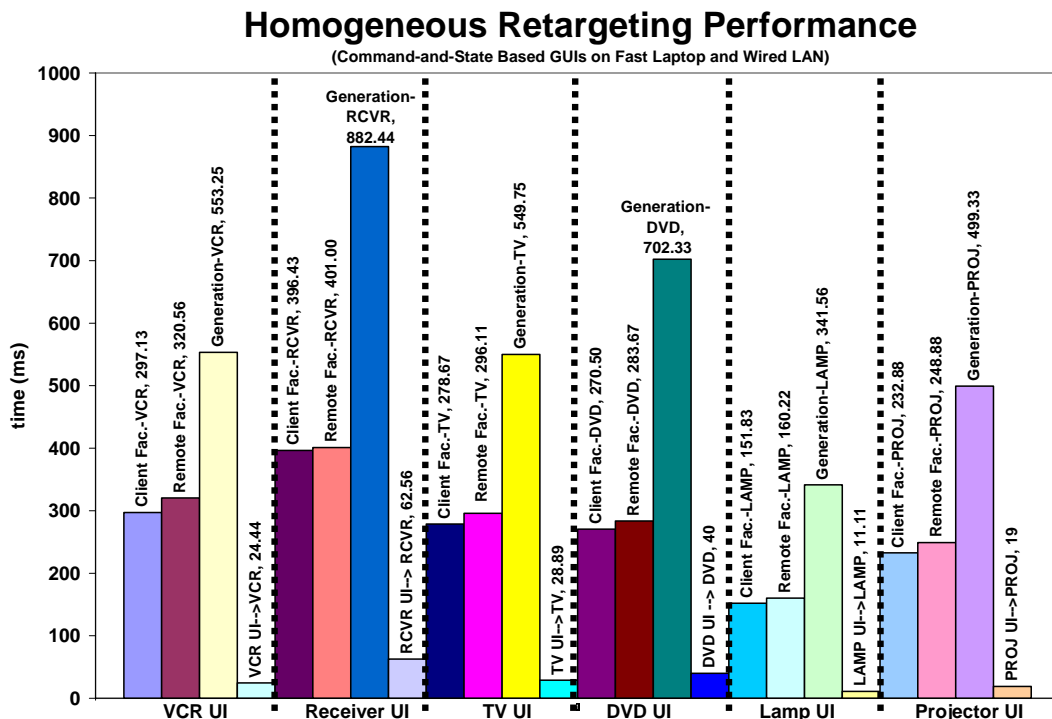


Figure 53. Homogeneous retargeting of command-and-state GUIs using the fast laptop and wired LAN connection.

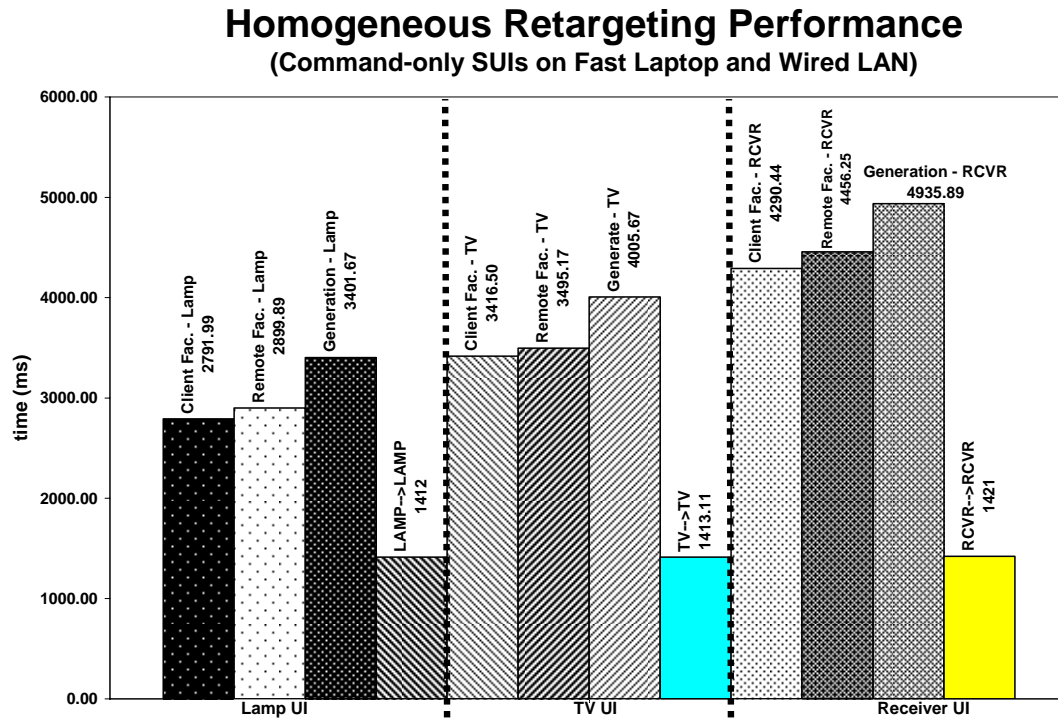


Figure 54. Homogeneous retargeting of command-only SUIs using the fast laptop and wired LAN connection.

In all cases, homogeneous retargeting is by far the fastest approach. Consider the task of deploying a receiver command-only GUI. Homogeneous retargeting is nearly thirty times faster than the client-factory approach. Deploying the command-and-state based GUI using homogeneous retargeting is over six times faster than the client-factory approach.

Notice that the speech-based retargeting times are over one second long, whereas, the GUI based times are only fractions of a second. It appears that even with retargeting, the SUI deployment times are still significantly greater than the GUI based times. Part of this continued difference is due to the fact that we consider deployment to be complete only when the system provides a user with the ‘start talking’ notification. We stop counting time only when a user can begin to use the SUI—i.e. speak commands.

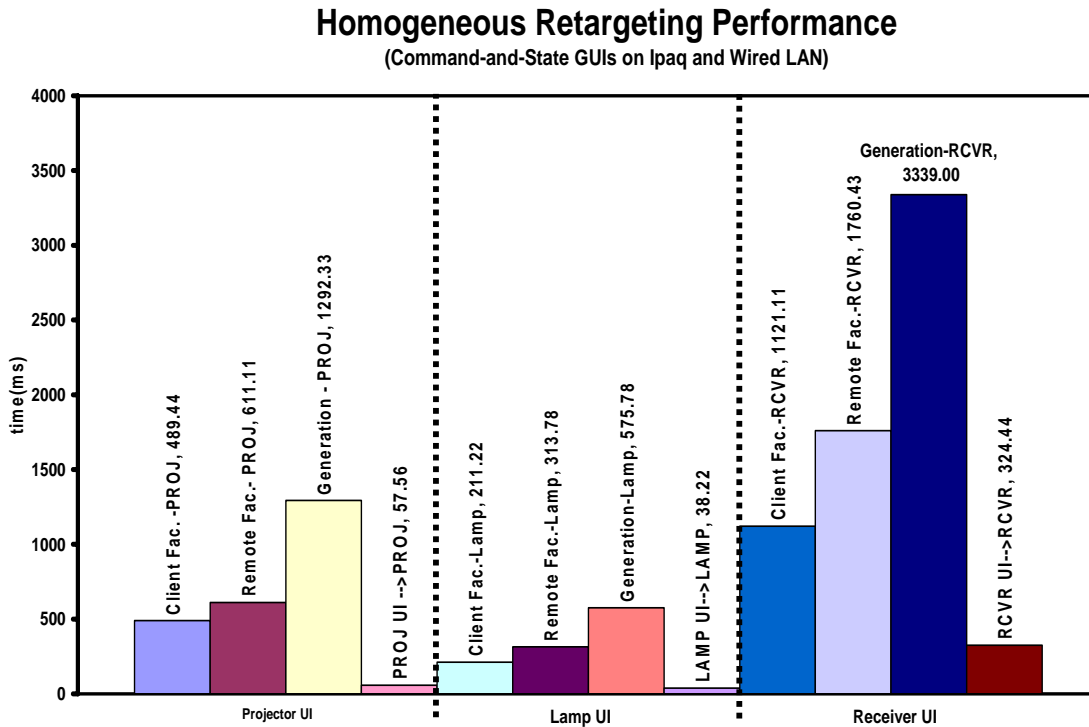


Figure 55. A graph comparing the homogeneous retargeting times to the corresponding times of competing approaches (using the Ipaq and a 100Mbps connection).

Figure 55 shows the homogeneous retargeting times for the Ipaq using the wired LAN connection. Here, the retargeting times were not as dramatically greater than the client-factory approach as seen with the laptop. As implied by this result and also Figure 48, remapping buttons and widgets is more demanding on Ipaq than the laptop. These Ipaq's times, however, are still multiple times faster than the client-factory based times. Retargeting the receiver user-interface, for example, is nearly 3.5 times faster than client-factory based times.

Though the above results show that homogeneous retargeting is very fast, it still has the earlier mentioned limitation of not supporting different source and target device programming interfaces. As a result, it can only support one of the five identified task transitions. In particular, it supports the 'turn on lights in a presentation room' transition by supporting LAMP UI→LAMP. The other four involve retargeting a user-interface of one device to a device of another type—that is, the types of the target and source devices

are not homogeneous. Homogeneous retargeting cannot, for example, support ‘setting up the projector in a presentation room’ since it cannot retarget a TV, VCR, lamp, or receiver user-interface to a projector. It is thus important to support heterogeneous retargeting.

4.4.3.2 Heterogeneous Retargeting

Using ObjectEditor (our mechanism) we considered several cases of heterogeneous retargeting. We produced such cases from the logs by extracting each instance where a user changes from one device to another. These device transitions appropriately provide the needed cases since each instance offers a source user-interface (from the previous device) and a new target device. As in the above evaluations, we only looked at the logs of users who owned the types of devices that we networked.

Using the imaginary presenter, we produced an additional set of device transitions. Recall that we assume that the presenter’s client has a set of source user-interfaces of some home devices available. Thus, we gathered all combinations of transitions from a home device to a conference room device. Overall, we produced a total of nineteen heterogeneous retargeting cases.

Figures 56 and 57 compare the non-caching based retargeting times to the times of the alternate approaches using the fast laptop with a wired LAN connection. Figure 58 makes a similar comparison for command-and-state based user-interface using the Ipaq.

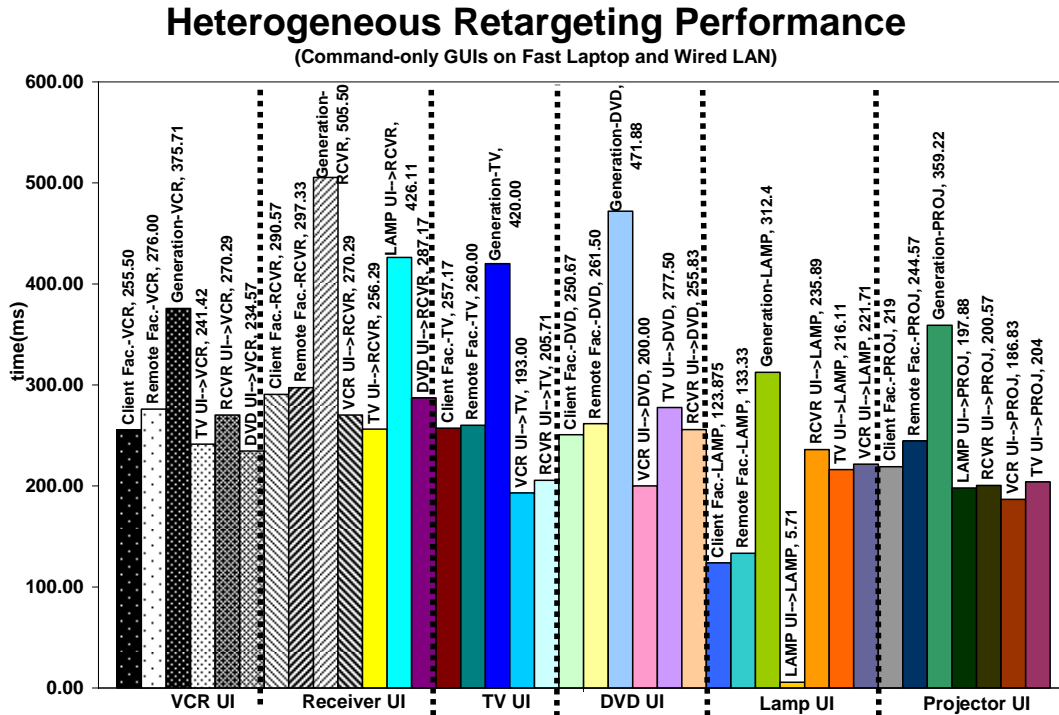


Figure 56. Heterogeneous retargeting of command-only GUIs vs. competing approaches (using the fast laptop and a wired LAN connection).

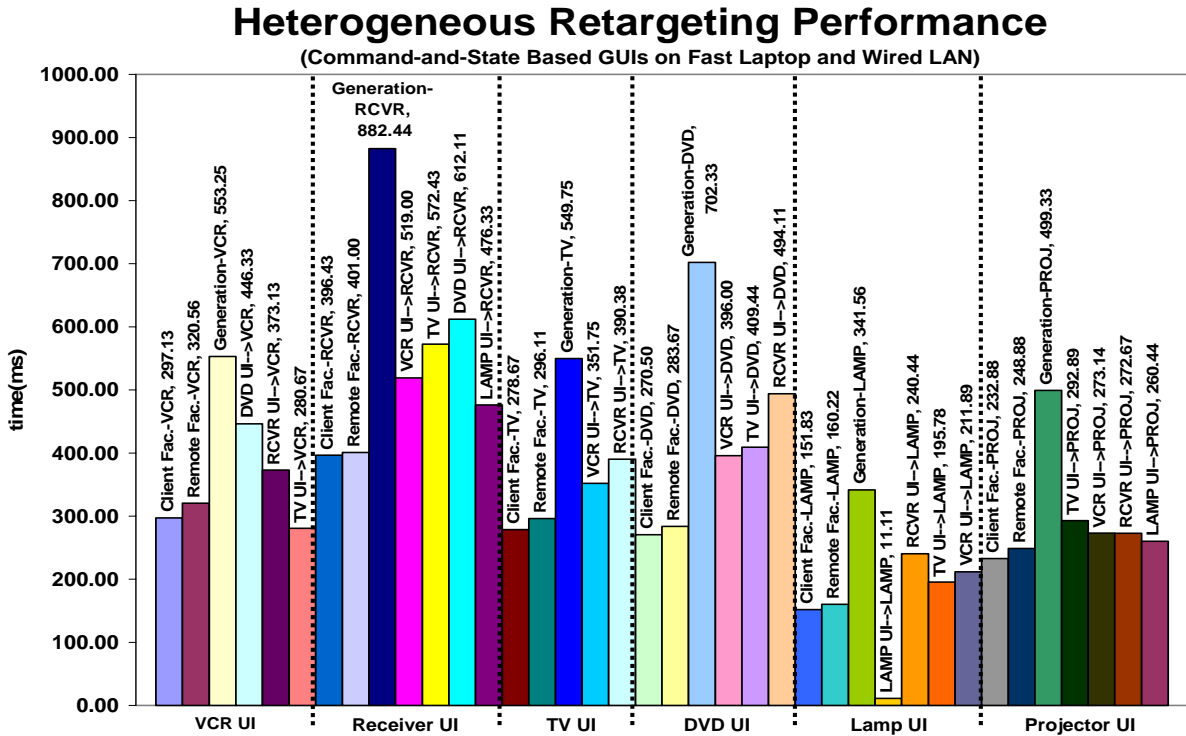


Figure 57. Retargeting times of command-and-state based GUIs vs. the corresponding times of competing approaches (using the fast laptop and a wired LAN connection).

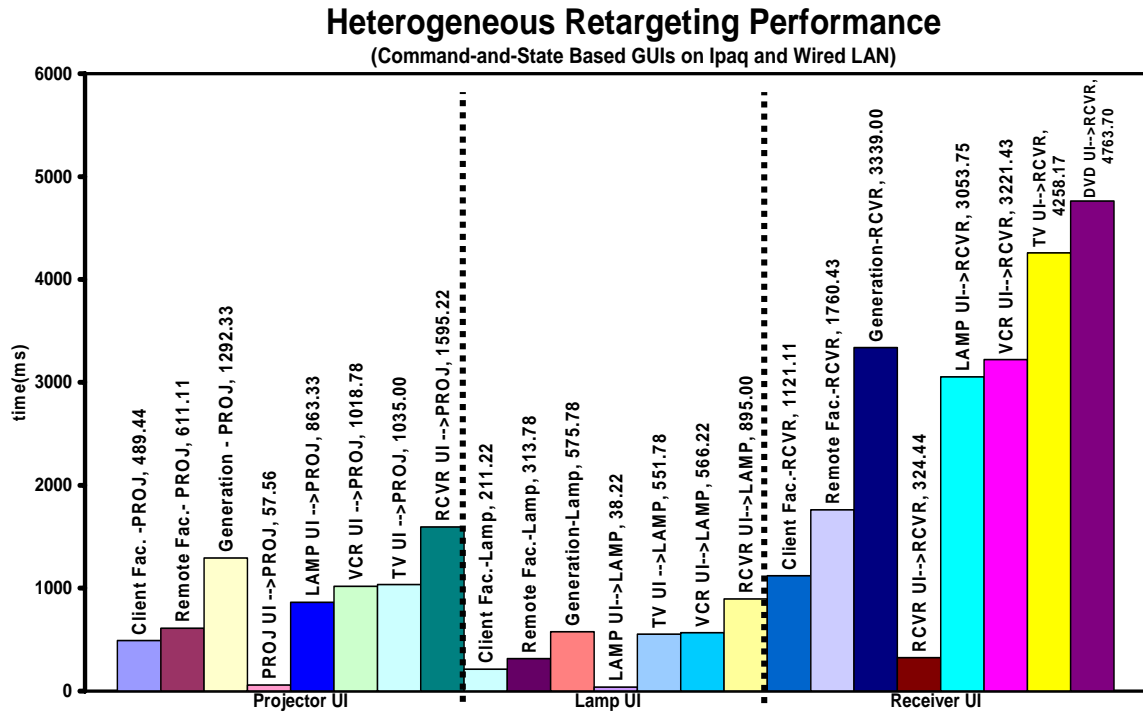


Figure 58. Retargeting times of command-and-state based GUIs vs. the corresponding times of competing approaches (using the Ipaq and a wired LAN connection).

The data shows some interesting results. For the case of retargeting command-only user-interfaces on the laptop (Figure 56), twelve of the nineteen cases yield times that are below the corresponding client-factory based times. Several of the other cases yield times that were relatively close to the client-factory based times. The retargeting times for the command-and-state based user-interfaces were not as promising. On the laptop, only one of the nineteen cases (TV UI → VCR) yields a retargeting time that is lower than its corresponding client-factory based time. The TV and VCR are our two most similar devices in terms of the commands and properties that they offer—hence this exception. On the Ipaq, none of the eleven heterogeneous retargeting cases that we considered yield times that are below the corresponding client-factory based times.

A likely reason for the low performance when including state is based on the processing involved in dealing with properties. As implied by our retargeting algorithm, supporting state (without caching) involves dynamically performing time consuming processes like searching getter and setter method signatures for a device's property names and type checking. Retargeting command-only user-interfaces does not involve such processes, hence the shown benefits.

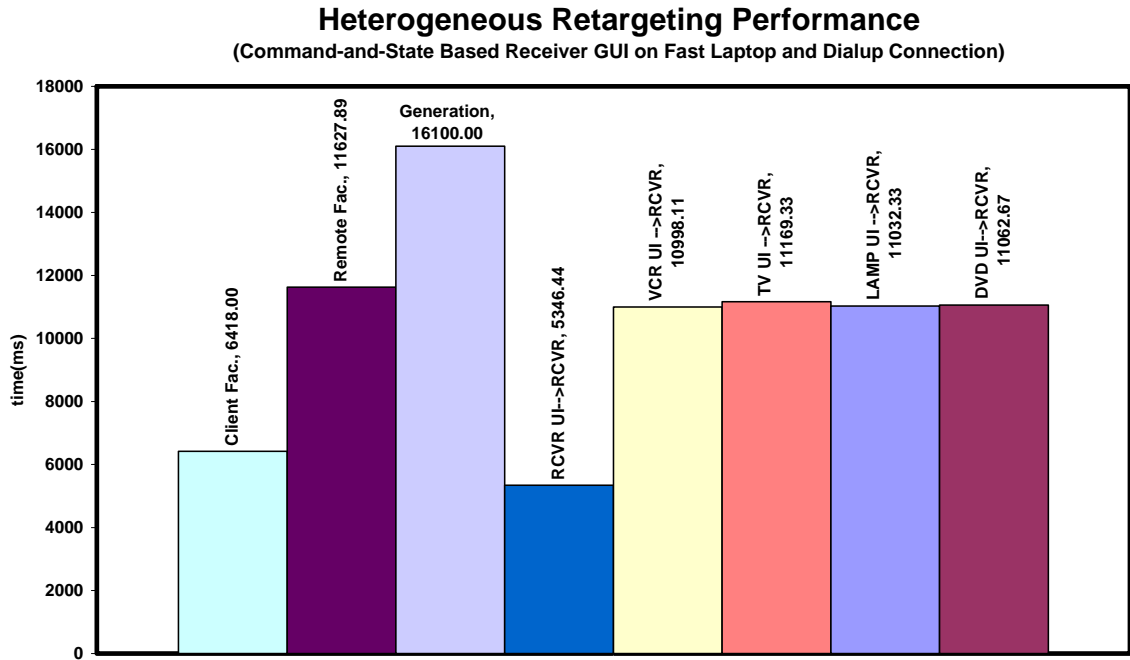


Figure 59. Retargeting times of the receiver command-and-state based GUI vs. the corresponding times of competing approaches (using the fast laptop and dialup connection).

To see the effects of a slower network on retargeting times, we also tested a subset of the retargeting on the laptop using the dialup connection (Figure 59). For this subset, we selected the cases in which the receiver is the target device. Recall from Figure 32 that we used the receiver to measure command-and-state based GUI deployment times of the three competing approaches under the dialup connection.

The results show that moving from wired LAN down to dialup speeds increases retargeting time by an approximate factor of twenty. One particularly interesting result is that the retargeting times of the heterogeneous retargeting cases plateau regardless of the chosen source device's complexity. Given that the corresponding wired LAN times did

not plateau (Figure 57), the occurrence here implies that the latency of a slower network can drown out some of the time benefits of retargeting.

Given the limitations of the non-caching based approach, particularly when deploying state-based user-interface, we measured the corresponding cache-based retargeting times. Recall that caching avoids repeating the kinds of time consuming processes mentioned above. Figures 60 and 61 compare the cache-based retargeting times to the times of the alternate approaches using the fast laptop with a wired LAN connection. Figure 62 makes a similar comparison for command-and-state based user-interface using the Ipaq.

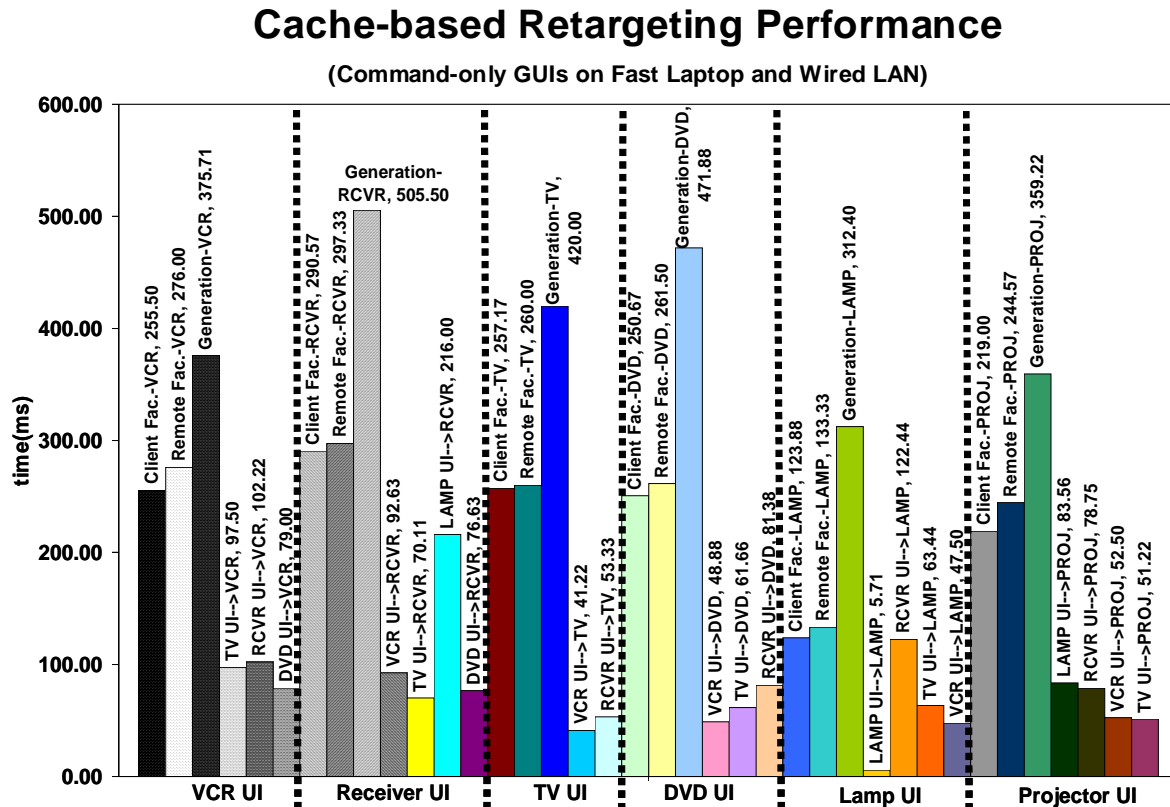


Figure 60. Cache-based retargeting times of the command-only GUIs vs. the corresponding times of competing approaches (using the fast laptop and wired LAN).

Cache-based Retargeting Performance

(Command-and-State Based GUIs on Fast Laptop and Wired LAN)

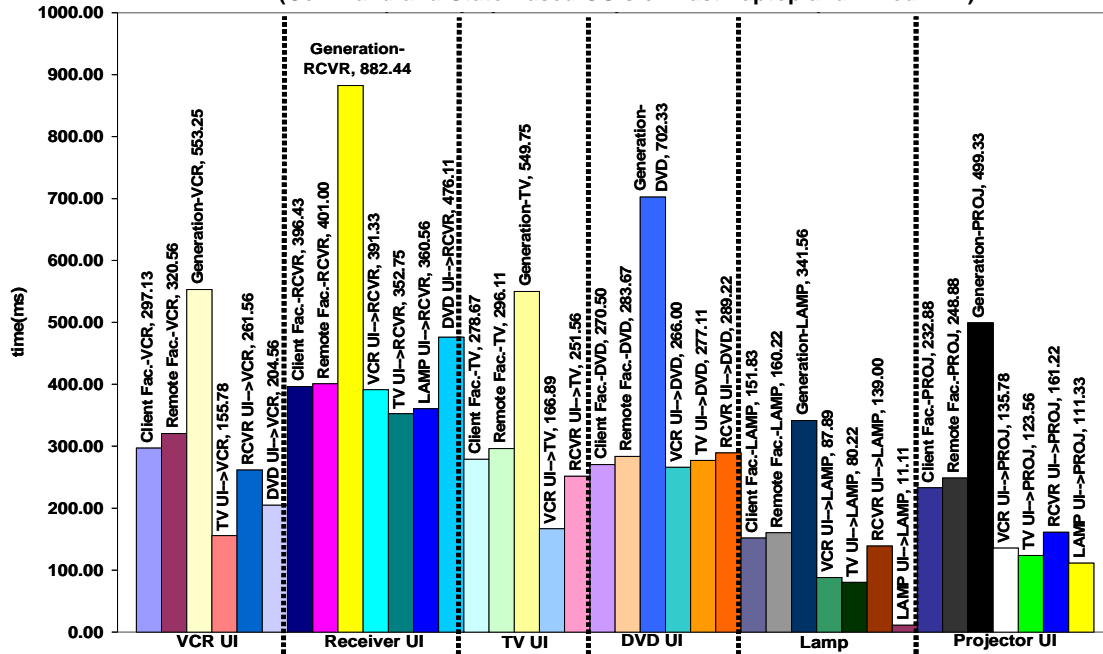


Figure 61. Cache-based retargeting times of the command-and-state based GUIs vs. the corresponding times of competing approaches (using the fast laptop and wired LAN).

Cache-based Retargeting Performance

(Command-and-State Based GUIs on Ipaq and Wired LAN)

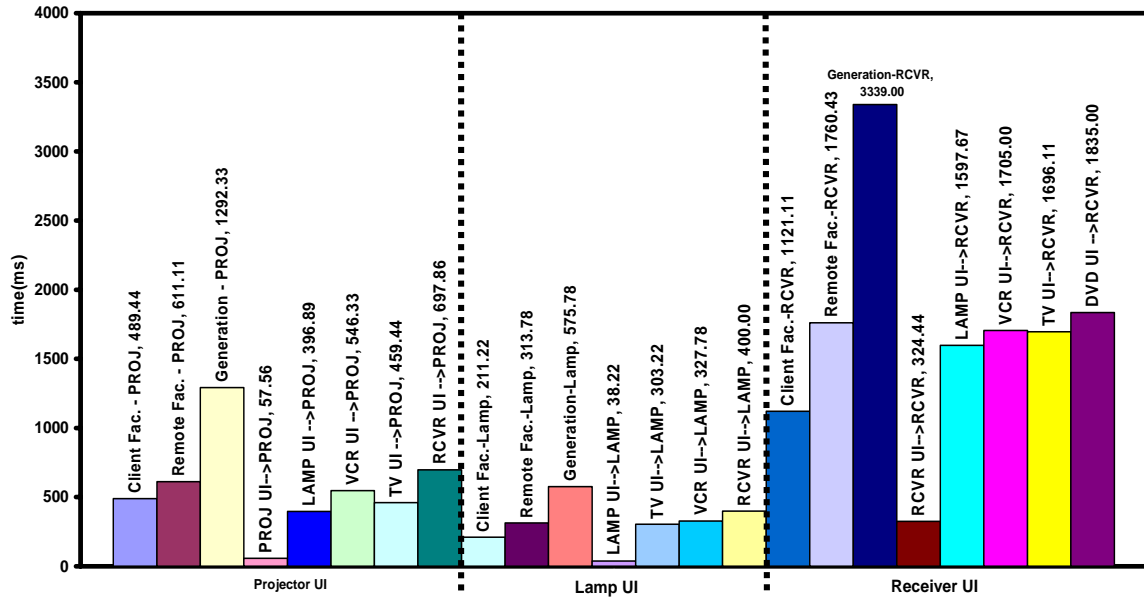


Figure 62. Cache-based retargeting times of the command-and-state based GUIs vs. the corresponding times of competing approaches (using the Ipaq and wired LAN).

The results show that caching offers major benefits. For the case of retargeting command-only user-interfaces on the laptop (Figure 60), all of the nineteen heterogeneous retargeting cases of Figure 56 yield times that are now below the corresponding client-factory based times. For command-and-state based user-interfaces recall that the non-caching case only offers one promising case. Caching improves this number to sixteen. Two of the three other cases, (TV UI→ DVD) and (RCVR UI→ DVD), have retargeting times that are respectively only two and three percent longer than their corresponding client-factory times. However, the other case, (DVD UI→ RCVR), is seventeen percent longer than its competing client-factory time. The DVD player and receiver are our two most complex and dissimilar devices.

In general, the benefits of cache-based retargeting on the Ipaq are not as significant as on the laptop. Only two of the eleven heterogeneous retargeting cases we tested yield times that are lower than their respective client-factory based times. Seven of the eleven cases, though, are faster than using the remote-factory approach—a major improvement over generation times.

As done with non-caching based retargeting, we tested a subset of the cases on the laptop using the dialup connection and caching support. We used the same subset of receiver-based cases from the earlier measurements so that we could make appropriate comparisons. Figure 63 shows the measured times from the experiments. Unlike the non-caching based results, all the cache-based retargeting times are lower than the corresponding client-factory based times. This result is even true for the (DVD UI→RCVR) transition, which we found to be much slower than client-factory deployment when using the wired LAN. It is currently not clear why this benefit occurs in the dialup case and not the wired LAN case.

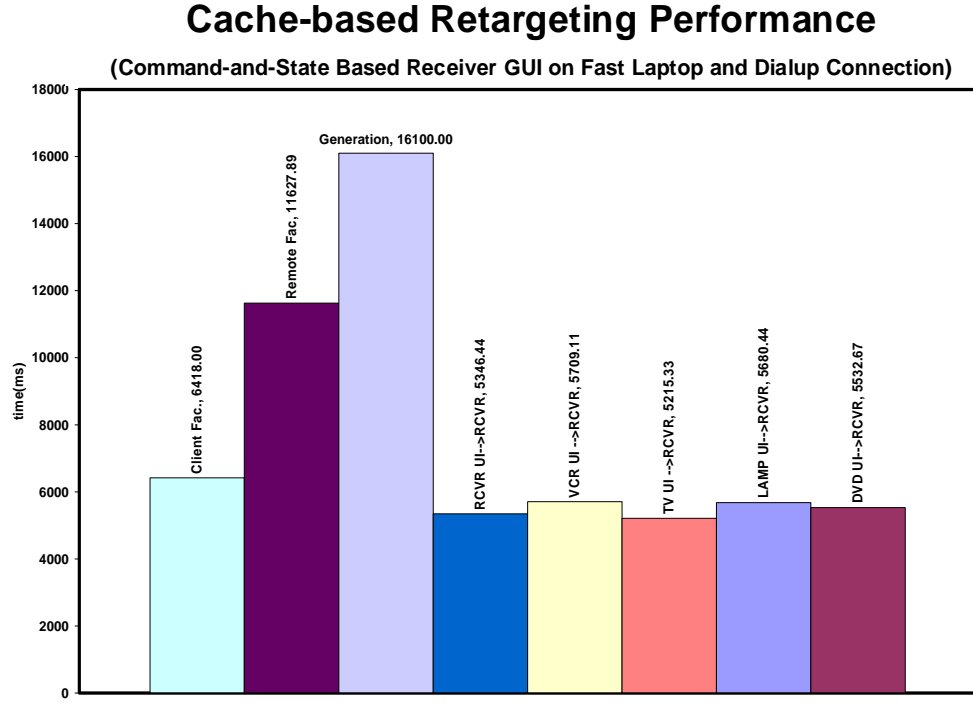


Figure 63. Cache-based retargeting times of the receiver command-and-state based GUI vs. the corresponding times of competing approaches (using the fast laptop and dialup connection).

4.5 Conclusion

The chapter presents the idea of user-interface retargeting. We identified several levels of retargeting a system can support and implemented some of them into a SUI and GUI generator. The GUI generator (ObjectEditor) supports the particularly important ability to retarget between devices of different types whose user-interfaces consist of buttons and primitive type based widgets. This ability is not currently supported by any existing generator. To efficiently support it, our implementation addresses the fastest user-interface selection and approach selection issues raised by such retargeting. We address them both using *regression-based source-device prediction*. As described earlier, this approach has a problem of long search times since it involves traversing the programming interfaces of possibly many source devices to gather information needed to make predictions. To optimize this process, the generator additionally supports cache-based retargeting.

Using our retargeting mechanisms, we prove the *Time-Efficient Generator Hypothesis*: it is possible for SUI and GUI generators to have deployment times that are often as good as or noticeably better than the inherently fastest approach of locally loading device-specific user-interface code. Based on our experiments we have found situations where retargeting times are better and worse than client-factory based times. More specifically, our experimental results show the following:

- *Regression-based source-device prediction* successfully addresses the fastest user-interface selection issue. In the experiments we ran, the generator always predicts the absolute fastest GUI even when the difference between the times two potential source user-interfaces offer is only one percent.
- *Regression-based source-device prediction* is also successful in deciding whether to generate or retarget. The experiments imply that retargeting is always faster than generation given that none of the many retargeting cases that we considered yield times that are greater than their corresponding generation time.
- Regardless of the kind of user-interface deployed (SUI vs. GUI) or the client's processing power (laptop vs. Ipaq), homogeneous retargeting times are an order of magnitude lower than their corresponding client-factory times.
- For heterogeneous retargeting, the times depend on whether: (a) the source user-interface is state-based and (b) caching is turned on.
- With no caching and command-only user-interfaces, most of the retargeting cases offer times that are lower than the client-factory based times.
- For command-and-state based user-interfaces, however, nearly all of the non-caching based retargeting times are significantly above their corresponding client-factory based times.
- Turning on caching, drastically improves retargeting time. On the laptop, almost all of the retargeting cases we considered have times that become lower than the client-factory based times after activating caching.

- In the Ipaq's case, the cache-based retargeting times are closer to the remote-factory based times than those of the client-factory. These times, however, are still a drastic improvement over generation times.

Chapter 5: History-based Generation

In the previous chapter, we prove the *Time-Efficient Generation Hypothesis* using the idea of user-interface retargeting. Here, we present history-based generation, which is another approach for addressing this hypothesis. Unlike user-interface retargeting, history-based generation avoids generating user-interfaces that support an entire device's functionality. Instead, it presents just the content a user needs in a user-interface, based on the user's past behavior. It supports the principle that the less content a user-interface will contain, the less time it should take to generate the user-interface. Hence, the assumption is that the content a user needs for a device's user-interface is generally less than the content needed in presenting the device's entire capabilities. Figure 64 illustrates a scenario where this content assumption is true. The generated GUI in the left contains all of the 42 buttons found on our networked receiver's remote control. The one on the right just contains the 10 buttons that the owner (the author) typically needs. Other commands beyond the ten shown have actually been invoked on the receiver during its history and are thus required in theory. However, these commands were only used during the initial setup of the receiver after it was purchased. Most of these commands, which include 'test tone', 'center down', and 'center up', were used to calibrate the speaker volume settings for the living room containing the receiver.

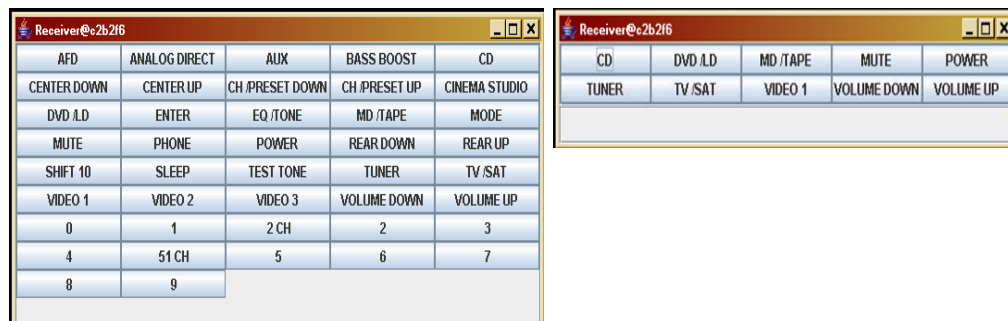


Figure 64. An entire receiver GUI (left) vs. a receiver GUI containing all of the commands the owner (author) typically needs (right).

The content assumption implies that history-based generation could also be used to address the problem of limited screen space offered by mobile computers when displaying GUIs. To illustrate this problem, consider the ObjectEditor-generated command-only GUIs for our six networked devices (Appendix A). They only require one screen on the laptop's 14 inch display. On the Ipaq pocket PC, which has a 3.8 inch display, the lamp GUI is the only case that requires a single screen. The GUIs for five other devices span at least two screens (Table 11). Consequently, they will force users to tediously scroll and tab through multiple screens to find buttons. When moving from pocket PCs to cell phones, which generally have even smaller sized screens (Figure 6), the tediousness of this scrolling increases. It is thus our *Space-Efficient Generation Hypothesis* that history-based user-interfaces can consume significantly fewer screens than their corresponding full device user-interfaces.

Device	# of Ipaq Screens for Full GUI
Receiver	3
DVD Player	3
TV	2
VCR	3
Projector	2
Lamp	1

Table 11. Number of screens consumed by each device's command-only GUI.

In the next section, we describe our history-based generation mechanisms in detail. Using these mechanisms, we evaluate how well we can prove the *Time-Efficient Generation* and *Space-Efficient Generation* hypotheses. Finally, we present our conclusions.

5.1 Approach

In forming our specific history-based generation approach, we focused on two important requirements:

- 1) *Automation*: The number of devices that a user periodically interacts with can be large. Thus, an approach should not be manual, requiring users to explicitly teach a generator which commands to filter out from the full user-interface of each device they use. It should offer automation by monitoring a user's interactions with each device over a period of time and then predicting needed commands.

- 2) *Fast Activation*: A user should not have to interact with a device for a long period of time before an approach can actually begin supporting the user's tasks with the device. When given a history-based user-interface for the device, the user should not have to constantly revert to the device's full user-interface due to missing commands. Further, the history-based user-interface should not contain many extra commands that are unneeded by the users.

In theory, it appears that a tradeoff exists between the two requirements. The manual approach implies fast activation since it can support a user's tasks with a device without observing the user actually use the device. Basically, a generator can begin deploying history-based user-interfaces directly after the teaching process. This teaching process, however, conflicts with the automation requirement. Automation, on the other hand, incurs a training period.

From analyzing the interaction logs mentioned in the previous chapter, we found that in practice, the two requirements can be easily met. The logs show that it can take a short period of usage time with a device for a user to invoke all of the commands needed in his/her common tasks involving the device. Figure 65 shows this observation using data from logging study participants. For a large collection of different kinds of devices, it shows for each device, the number of usage days required by its owner to invoke all of the commands he/she needs in common tasks involving the device—we call these commands *common-tasks-commands*. To gather these values, we individually asked the participants to provide us with a list of tasks they commonly perform with their devices (e.g. watching TV and listening to music). We then asked them to look at each of their remote controls and list the commands needed for each of their associated tasks. The combination of commands of a particular device, across all lists of commands provided by a participant, defines that participant's *common-tasks-commands* for the device. Given a device's *common-tasks-commands*, we searched its owner's log to find the number of usage days required to invoke them. When searching the logs, we found that some participants forgot to list some remote control commands they actually use during their tasks. We simply added these command names to the command list of the appropriate task.

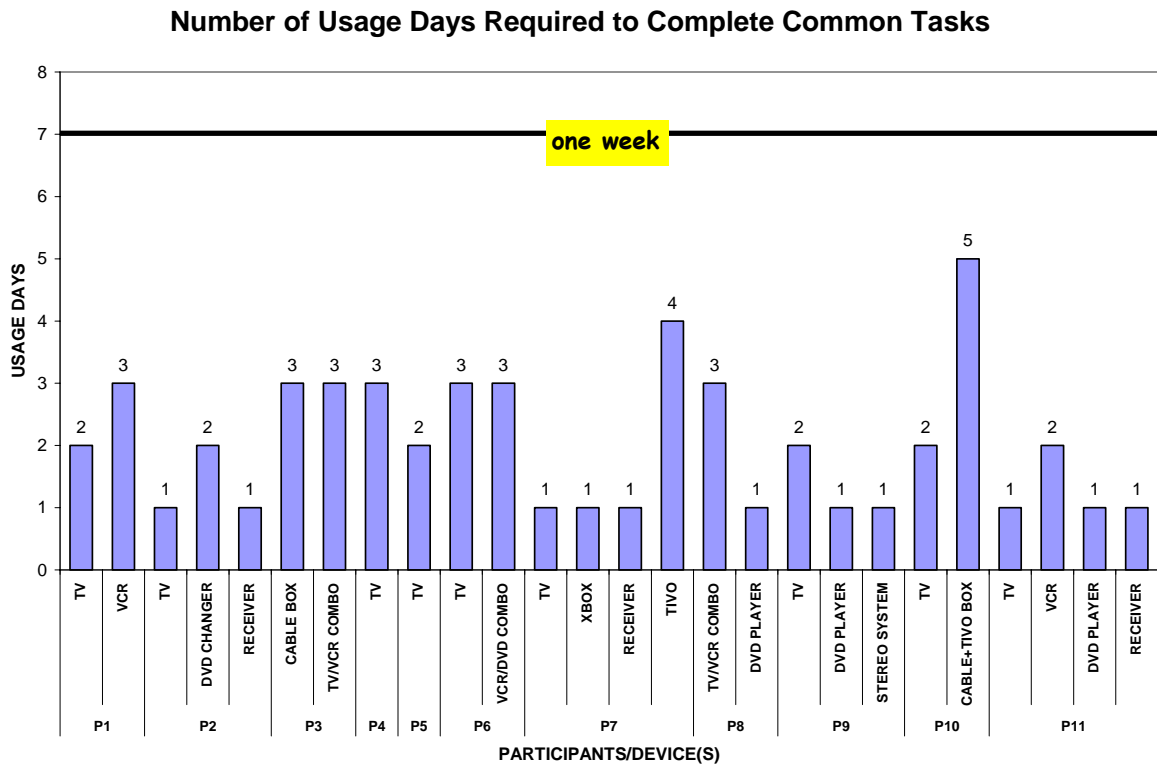


Figure 65. Number of usage days required for the participants to complete their common tasks on their respective devices.

To summarize, the data shows that the participants required less than a week of normal usage with most of their respective devices in order to invoke all of their common task based commands. In fact, for most cases (17 of 26 total devices), only one or two days of usage were required. Figure 65 omits the usage data for some of the participants' devices that were actually used during their week of logging. For example, it leaves out participant-5's (P5) stereo system and DVD player data. The reason is that a week of logging was not enough to capture all of her *common-tasks-commands* of these devices. To illustrate, P5 listed the CD1, CD2, and CD3 commands of her stereo system as being needed in her 'listening to music' task. These commands play the CD in a given slot (1-3) in the stereo system. During logging, she only played the CD in slot 1 by pushing CD1, thus, the logging mechanism did not capture the CD2 and CD3 buttons. P5 did mention that all CD slots in the stereo contained a CD throughout the week, but she had only been interested in listening to the CD in slot 1. In fact, she further stated that: (a)

she had not listened to the CDs in slots 2 and 3 “for a long time” and (b) whenever she needed to insert a new CD of interest in the system, she would replace the CD in slot 1. Thus, technically, her week of logging did capture the commands she would need for a substantial amount of time.

Given the device usage data presented above, an approach that allows us to meet our two requirements is for a generator to automatically: (a) record the commands the user invokes on a device over a short period and then (b) create a user-interface consisting of just those recorded commands. In fact, the data suggests that for a variety of devices and users, a generator can begin deploying history-based user-interfaces after a week of logging. We imagine that there could be some cases in which one week is not enough. Therefore, the generator should have a fallback mechanism for efficiently reverting to a device’s full user-interface if a history-based one is incomplete. Since a device’s full user-interface contains all of the commands of any of its history-based ones, this fallback can use the retargeting techniques we developed to quickly perform the switch. That is, it can covert the history-based user-interface (source) into a full one (target) by simply adding the unrepresented device commands.

5.2 Evaluation

We extended our SUI and GUI generators with the necessary functionality to evaluate the generation time and screen space efficiency of our approach. In full generation, a generator dynamically extracts a device’s commands at interaction time and represents all of them in a user-interface. Under history-based generation, the generator still extracts device commands. However, it presents a user-interface consisting of only the commands found the log.

Ideally, we could perform our evaluation by using each of our study participant’s log to generate actual history-based user-interfaces and then take deployment time and screen space measurements. This process, however, requires networking over twenty devices of the participants in the manner followed in our earlier performance experiments (Section 3.1). The reason is that the participants’ devices were of varying brands and generally

offered unique sets of commands. None of the participants, for example, had TV remote controls with the same set of buttons.

Even beyond having to network all devices, such uniqueness in device commands also requires handcrafting the GUIs and SUIs needed to make comparisons to the predefined approach. Due to lack of time and the overhead involved, we were unable to implement the described setup. To still make a general quantitative evaluation, we deployed history-based user-interfaces using the author's logs and used the collected measurements to make estimations for the other participants' cases. Recall that it is the author's TV, VCR, receiver, and DVD player that we networked for the performance experiments described in the previous chapters. It is his interactions with these four devices that we logged in our study.

5.2.1 Generation Time Efficiency

For the four devices, Figures 66 and 67 compare their history-based GUI generation times to their matching client-factory, remote-factory, and full-generation based times. Respectively, the two figures make these comparisons on the laptop and Ipaq using a wired LAN connection.

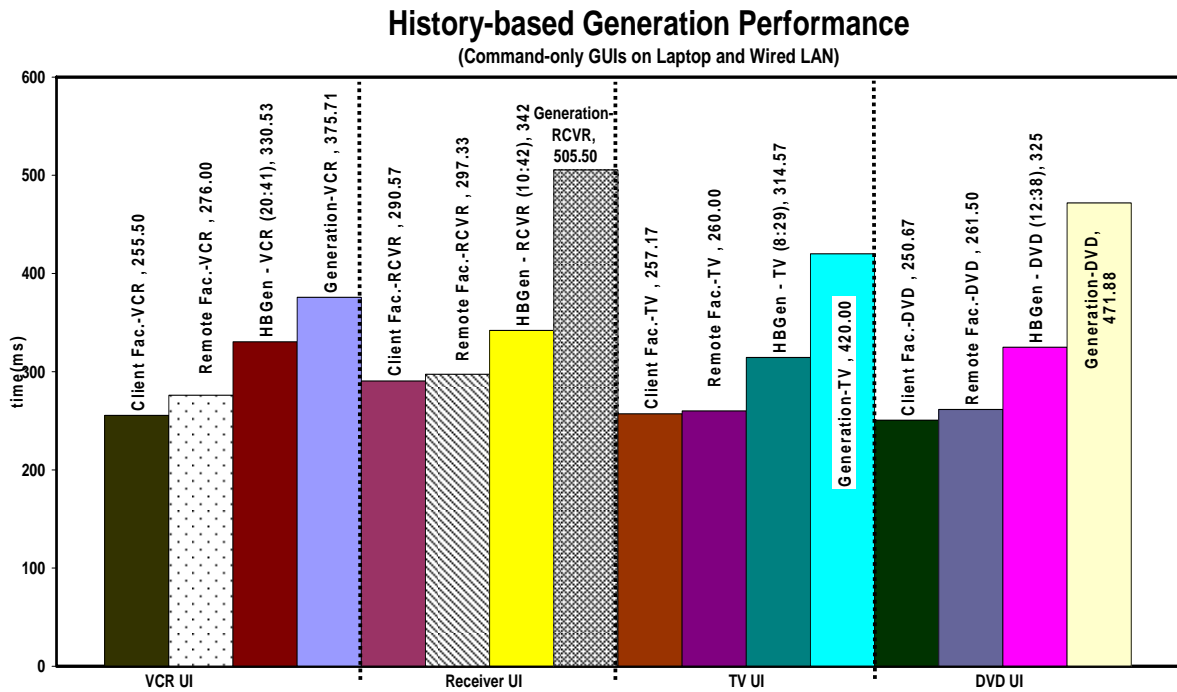


Figure 66. History-based GUI generation performance using the laptop and wired LAN connection.

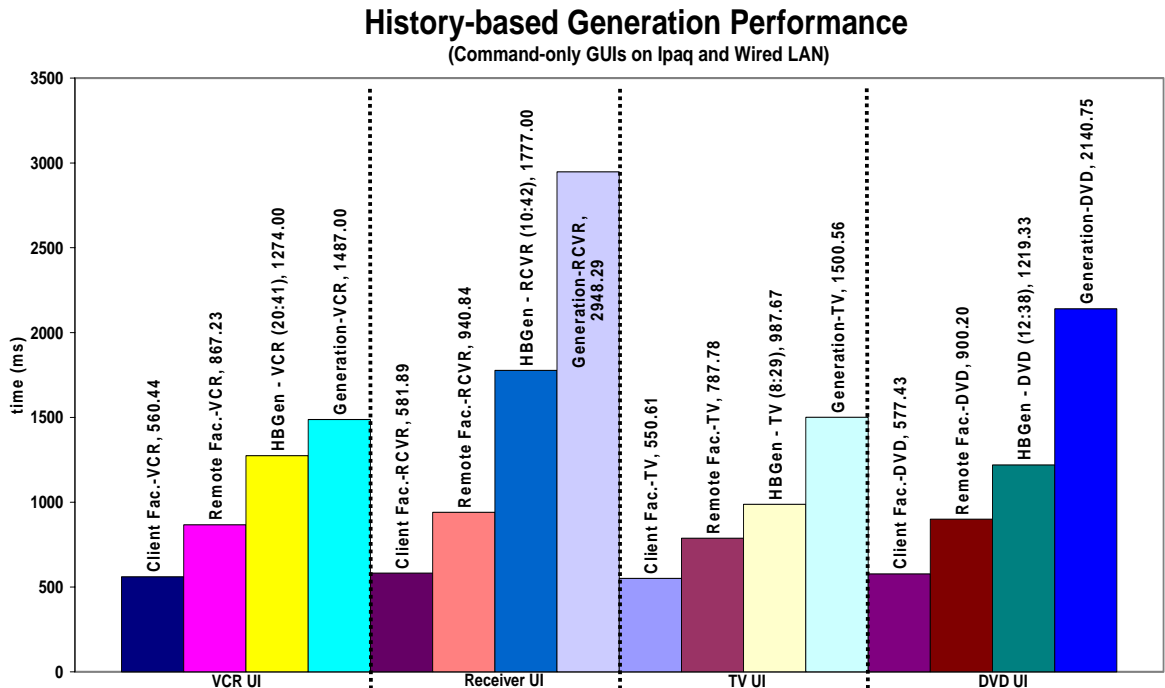


Figure 67. History-based GUI generation performance using the Ipaq and wired LAN connection.

The results show that even with more than half of each device's commands being filtered from its history-based GUI (Table 12), none of the GUIs have generation times that are lower than factory deployment times. Most cases, however, have significantly lower times than their corresponding full-generation times. Generating the receiver's history-based GUI, for example, eliminates 32% and 40% of full generation time on the laptop and Ipaq respectively.

Device	Ratio: # buttons displayed to total	Percentage of Commands Omitted
Receiver	10:42	76%
DVD Player	12:38	68%
TV	8:29	74%
VCR	20:41	51%

Table 12. A summary of command filtering amounts for each device.

We also evaluated history-based SUI generation performance on the laptop. Figure 68 compares the history-based generation times of the four devices' SUIs to the corresponding times of other approaches. The results for SUIs show a smaller percentage difference between the history-based and full generation times than in the GUI case. History-based SUI generation for the receiver, for example, is only 20% faster than full generation. Unlike the GUI case, however, all the history-based generated SUIs have deployment times that are lower than their competing client-factory times. This is even true for the VCR's history-based SUI which contains the most number of commands. This success is likely due to the fact that there is a smaller difference between client-factory and generation based deployment times for SUIs than GUIs. Hence, the client-factory times are easier to reach.

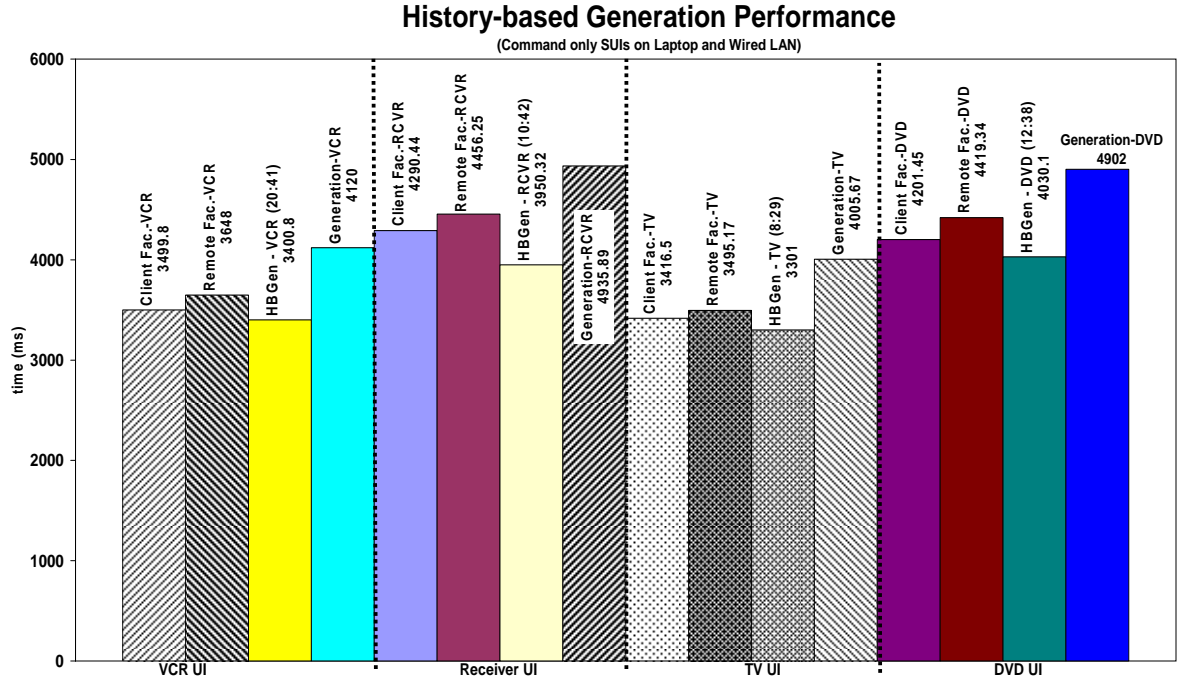


Figure 68. History-based SUI generation performance using the laptop and wired LAN connection.

Using the author's results presented above and a projection of the number of commands each history-based user-interface of the other participants would contain, we can make a more general evaluation about the performance of our approach. Without having to actually deploy each user-interface, we can discover its command total by counting the number of unique commands found in the log of its associated participant-device pair (Table 13). Given all user-interfaces, we found this number to span 1 to 32 commands. P7's history-based TV user-interface, for example, would only require a power button. Also, P2's history-based receiver user-interface would just require volume up and down buttons. Most of the participants' history-based user-interfaces, however, would not have such low command totals. In fact, 17 out of the 22 total history-based user-interfaces would contain at least 10 commands. Recall that the author's history-based TV GUI contains only 8 commands, yet its generation time still does not meet its corresponding client-factory based times. It is thus very likely that most of the other participants history-based generated GUIs will not have deployment times that are better than their corresponding client-factory based times. These GUIs, however, will certainly have deployment times that are better than their corresponding full generation times. The

reason is that they all require fewer commands than their corresponding full GUIs—there are no cases in which a participant requires all of a device’s available set of commands in performing his/her common tasks.

Participant	Device	# of UI commands
P1	TV	17
	VCR	26
P2	TV	18
	DVD CHANGER	17
	RECEIVER	2
P3	CABLE BOX	24
	TV/VCR COMBO	10
P4	TV	20
P5	TV	17
P6	TV	23
	VCR/DVD COMBO	23
P7	TV	1
	XBOX	11
	RECEIVER	5
	TIVO	32
P8	TV/VCR COMBO	26
	DVD PLAYER	6
P9	TV	10
	DVD PLAYER	16
	STEREO SYSTEM	3
P10	TV	3
	CABLE+TIVO BOX	27

Table 13. Number of commands required in each participant’s set of history-based user-interfaces.

For a more quantitatively-based prediction of history-based generation time savings, we can use the generation-time estimation function presented in the previous chapter (Section 4.2). This comparison involves calculating the function for both the history-based and full GUI of each participant-device pair. As inputs to the function, one calculation uses the number of *common-tasks-commands* a user requires on a given device while another uses the number of commands on the device’s remote control. These inputs correspond to the number of buttons on the history-based and full GUI for the participant-device pair, respectively. Unfortunately, we did not count the number of buttons found on our participants’ remote controls during the time of our logging study since the values were not needed for our original reasons for performing the study. Thus, we are currently unable to offer any quantitatively-based predictions of history-based generation time savings over full generation using our participants’ data.

For SUIs, we expect a similar success as found with the author's data. Recall that even the author's history-based VCR SUI, which contains 20 commands, had a generation time that was lower than the matching client-factory based time. Most of the other participants' history-based user-interfaces (15 out of 22) would contain 20 or fewer commands and would thus yield similar results.

5.2.2 Screen Space Efficiency

We are only concerned with the screen space efficiency of our approach on the Ipaq since a single laptop screen can display GUIs containing very high numbers of buttons. For the author's four history-based GUIs, we counted the number of Ipaq screens required to display them (Table 14). Three of the four GUIs require only one screen, compared to all four requiring at least two for the full GUIs. Figure 69 portrays this benefit by showing the history-based receiver GUI on a single Ipaq screen.

Device	# of Screens Consumed	
	Full UI	His.-Based UI
Receiver	3 (See Figure 5)	1
DVD Player	3	1
TV	3	1
VCR	2	2

Table 14. The number of Ipaq screens required for full and history-based GUI.

Most of other participants' history-based GUIs would also offer similar benefits as those of the author. The reason is that our generator fills an Ipaq screen with a maximum of 18 buttons, and most of the participants history-based GUIs require 18 or fewer buttons. Specifically, 14 of the 22 total cases would only require a single screen on the Ipaq. As mentioned earlier, we did not count the number of buttons found on the remote controls of each of the other participants' devices. However, by simply considering the complexity of their devices, we can say that all of the associated full GUIs would require more than one screen.



Figure 69. The receiver’s history-based GUI on a single Ipaq screen. With the available space from filtering buttons, the remaining buttons can be stretched to fill the screen.

5.3 Conclusion

This chapter presents the idea of history-based user-interface generation. Our specific approach is based the observation that it can take a short period of usage time with a device before a user invokes all of the commands needed in his/her common tasks involving the device. A generator can thus: (a) record the commands a user invokes on a device likely over a short period of time and then (b) create user-interfaces consisting of just those recorded commands.

We extended our SUI and GUI generators with the needed mechanisms to evaluate whether this approach can be used to prove our *Time-Efficient Generation* and *Space-Efficient Generation* hypotheses. Using the logged interaction data of several users and their different devices, we prove the former hypothesis within the scope of history-based SUI generation. History-based GUI generation, though significantly faster than full GUI generation, does not allow us to meet this hypothesis. It does allow us, however, to prove the *Space-Efficient Generation* hypothesis—i.e. most history-based generated GUIs require only one screen compared to two or more for full generation.

Chapter 6: Pattern-based Composition

The ideas presented thus far in this dissertation apply to both single and multiple device user-interfaces. The multi-device case also raises the additional issue of how devices are composed. As our discussion of related work shows (Section 2.2), current examples demonstrate different approaches to supporting such functionality.

Some infrastructures provide users with already programmed mechanisms for achieving desired compositions. Cougar and TinyDB, for example, are two high-level infrastructures that provide composers for performing multi-sensor queries. The two infrastructures, however, are limited to only supporting queries. Infrastructures have been built for generically supporting composition. ICrafter and Speakeasy are the two examples that exist today. Both of them, however, are low-level since they place much of the composition effort on their users and programmers. In our description of these two systems (Sections 2.2.5 and 2.2.6), we show that this burden is not small largely due to the combinatorics involved in flexibly supporting composition.

In summary, current infrastructures force us to choose between high-level support and composition flexibility. Specifically:

- 1) each existing high-level infrastructure supports composition semantics that no other high-level infrastructure supports.
- 2) each low-level infrastructure can flexibly support each of the existing composition semantics but requires a costly amount of effort from its users and programmers.

It thus our *High-level and Flexible Composition Hypothesis* that a new infrastructure can be built to overcome this existing tradeoff by meeting the two conditions below:

- 3) supports the composition semantics of existing high-level infrastructures.

- 4) provides higher-level support than all other infrastructures that can support all of these semantics.

In the next section, we present an overview of our approach to meeting the hypothesis. We then evaluate this approach by describing an implementation that we built to meet the two above conditions. Finally, we present our conclusions.

6.1 Overview

Our process for achieving a high-level and flexible composition framework involved three steps. The first step was to gather the specific composition examples supported by current infrastructures and abstract them into a set of operations. With a range of operations defining our target flexibility, our next step was to find out why existing systems cannot properly support them at a high-level. Given these reasons, we then derived a framework that would allow us to design and implement the necessary algorithms for supporting each operation.

We identified seven different abstract operations. Below, we motivate and describe them:

- *‘GUI Stack’ Operation* – Earlier, we described how Hodes’ System could vertically and horizontally stack the individually generated GUIs of a set of room lamps to form a single compound user-interface. This compound GUI allows a person to set the room’s lighting without tediously switching between individual lamps user-interfaces. Hodes’ System thus performs what we refer to as the ‘GUI Stacking’ operation, which stacks the individual user-interfaces of a set of devices into one. A movie watcher, for example, might perform the ‘GUI Stacking’ operation on a TV, DVD player, and receiver to avoid switching between their individual GUI while watching a movie.
- *‘GUI Merge’ Operation* – Recall that the individual GUIs of the author’s networked TV, DVD player, and receiver, each consume three Ipaq screens (Section 5.5.2). On a mobile device with a small screen, the movie watching stacked GUI mentioned above could present a scrolling problem that is similar to

tediously switching individual TV, DVD player, and receiver user-interfaces. Basically, this GUI could require the movie watcher to scroll back and forth over a significant amount of screen real-estate in order to access individual device buttons. More generally, the scrolling cost of stacked GUI can be quite high in compositions involving complex and/or numerous devices, such as the movie watching case.

To reduce scrolling, a ‘GUI stacking’ composer could support our history-based generation approach. That is, for each device’s panel, it could display the device’s *common-tasks-commands*—i.e. the commands a user typically invokes during common tasks involving the device. Based on our history-based generation evaluation (Section 5.2.2.), this approach should be quite successful in reducing the number of buttons to display. Using the author’s usage data, the movie watching stacked GUI would reduce from 109 to 30 buttons (spanning TV, DVD player, and receiver commands). Though significantly smaller than the full GUI, the one showing only *common-tasks-commands* of the three devices is still not optimal since the author only requires 19 commands when watching DVDs. Ideally, a compound GUI for watching a DVD should only display these 19 commands in order to minimize scrolling on small screens. We thus introduce the GUI merge operation, which creates such a compound GUI. This operation merges the operations of a set of devices to create task-specific compound GUIs. A user, for example, might want a ‘music listening’ merged GUI for a CD player and receiver. This GUI would omit space consuming commands such as those for burning a CD and setting the receiver to different radio channels.

- *‘Do All’ Operation* - Earlier, we described how an ICrafter composer could allow a user to turn several lamps on and off with a single action (e.g. button click). This functionality allows the user to avoid performing many actions on many individual lamp user-interfaces. We appropriately refer to this kind of functionality as the ‘do all’ operation, which invokes a command on each member of a set of devices that causes the devices to perform a similar action. Besides

lamps, a user could invoke a ‘do all’ operation on several clocks to add/decrease an hour during daylight savings time.

- *‘Do Sequence’ Operation*– Macros are very useful in many of today’s conventional computer applications. It can also be useful to invoke macro operations on a set of devices. Palm/Pocket PC IR programs, for example, can allow a person create a macro that automatically prepares a TV, DVD player, and Receiver for watching a movie. This macro could perform the following actions:

- 1) Turn on the TV
- 2) Set TV to DVD video input channel
- 3) Turn on the receiver
- 4) Set the receiver to DVD audio input
- 5) Turn on the DVD player
- 6) Open the DVD player’s disc tray

In spirit of the ‘do all’ operation, we refer to the functionally provided by a composer invoking such a macro as the ‘do sequence’ operation. A ‘do sequence’ operation could be applied on the same DVD and receiver to preparing them for music listening:

- 1) Turn on the receiver
- 2) Set the receiver to DVD audio input
- 3) Turn on the DVD player
- 4) Open the DVD player’s disc tray

- *Query Operation* - As Cougar and TinyDB show, it can be useful to perform queries on networked sensors that are distributed over an environment. Cougar, for example, allows a person to query various sensors at a specific location for their average rainfall measurements. It thus supports the query operation—i.e. the ability to search a set of devices to find those with attributes (e.g. location) of a specific value. We imagine several cases in which the general ability to query a set of devices can be useful. For instance, before going on vacation, users could query their homes for all the devices that are on. Using the references of those devices that meet the query, users can decide the appropriate action to save electricity—e.g. turn all the non-essential ones off.

- *'Data Transfer' Operation*- Another way to compose devices using their data values is to allow them to exchange these values over a network. Our earlier discussion of ICrafter and Speakeasy describes an example in which a composer provides the ability to transfer pictures in cameras to display devices for viewing. Similarly, Websplitter provides the ability to transfer webpage content (e.g. audio files) to presentation devices (e.g. sound systems). These systems thus support what we refer to as the data transfer operation, which allows information from a data producer to be transferred to a data consumer. This operation could be used to achieve many other useful compositions. For instance, it could allow the transfer of an atomic clock's time value to several other clocks after a power outage. It could also allow the transfer of the clock's time value to the internal clocks of newly purchased VCRs, TVs, and microwaves.
- *'Conditional Connect' Operation* – We identified a set of new kinds of examples that are very different from those currently demonstrated. Consider the ability to create an adhoc security system using a motion detector outside the front door of a house and various lights near the interior entrance. The devices are composed together so that when the motion detector senses motion, all of the lights, if off, are automatically turned on. Similarly, a DVD player could be composed to a telephone so that when the phone rings while a movie is playing, the movie is automatically paused. These examples rely on a general functionality provided by what we refer to as the 'conditional connect' operation. This operation automatically invokes one or more operations on a set of devices based on the state conditions of another set of devices.

In general, these operations represent a variety of ways to compose devices by their possibly many operations and/or data entities. Below, we describe the specific orientation required by a given operation:

- *'GUI Stack' and 'GUI Merge' Operations* – A data-oriented approach can provide a GUI consisting of only the state information of a group of devices. An

operation-oriented approach can only present their operations in a GUI. An infrastructure that supports both can present both state and operation information.

- *‘Do All’ Operation* – Operation-oriented by requiring access to the shared operations of a set of devices.
- *‘Do Sequence’ Operation* – Operation-oriented by requiring access to all operations of a set of devices.
- *Query Operation* – Data-oriented by requiring access to device attribute values.
- *Data Transfer Operation* – Data-oriented by requiring access to device data values.
- *Conditional Connect Operation* – Requires an infrastructure that supports both orientations since state and operations are required for conditions and matching events, respectively.

Supporting all operations at a high-level requires a data and operation oriented framework. A data-oriented framework views devices as collections of readable and/or writeable attribute values while an operation-oriented one views them as sets of operations. Table 15 classifies existing systems in terms of their orientation.

System	Orientation
Cougar	Data
TinyDB	Data
Hodes’ System	Operation
Websplitter	Data
ICrafter	Operation & Data
Speakeasy	Operation & Data
Palm/Pocket-PC IR Programs	Operation

Table 15. A classification of existing systems

It shows that Cougar, TinyDB, Hodes’ System, Websplitter, and Palm/Pocket PC programs are limited by their support of only one of the two orientations. ICrafter and Speakeasy, on the other hand, support both. Recall from our survey of the two systems that they support a programming interface based approach to composition (Sections 2.2.5

and 2.2.6). Devices must implement well-known programming interfaces that expose the ways in which they can be composed. These interfaces declare operations that devices must support in order to achieve the interfaces' associated semantics. Composers can then be programmed to support the semantics tied to specific interfaces, invoking the necessary interface-based operations on devices in order to meet their goals. Inherently, the interface-based approach is operation-oriented. Now given that interfaces only declare operations, how can the approach also support data-based operations? The insight provided by ICrafter and Speakeasy is that interfaces representing data-oriented semantics must declare standard operations that composers can invoke to access needed data. An interface for describing a device's data transfer composability would, for example, declare well-known operations for accessing the device's exchangeable data.

Although the interface-based approach supports both orientations, it still exhibits the important high-level support and flexibility tradeoff. This limitation arises in addressing the following question: How specific should interfaces be in exposing the composability of devices implementing them? In other words, how much detail should interfaces provide in describing the ways in which devices can be composed? Should a device, for example, implement a generic interface simply exposing its ability to exchange data, or should it implement a set of more specific interfaces describing the different ways it can exchange data with another device. As specificity increases, the kinds of semantics that can be supported within an infrastructure become more well-defined and thus more programmable (or automatable) using composers. In essence, the less ambiguity that programmers have in understanding the semantics supported by an infrastructure, the more tailored their composers can be in achieving those semantics. Infrastructures with high flexibility and interface specificity, however, require many interfaces since more interfaces are needed to separate the differences between many semantics. To illustrate, the more ways an infrastructure that is based on specific interfaces allows devices to exchange data, the more interfaces are needed to differentiate the different ways of data exchange. Given that greater automation requires more specific interfaces, supporting high flexibility and automation leads to a proliferation of interfaces and significant programming costs in writing associated composers. To composer programmers, a

tradeoff thus exists between flexibility and ease of use under the interface-based approach.

Keeping the level of specificity of interfaces low avoids this tradeoff since it allows for fewer (more generic) interfaces which leads to fewer composers to write. However, it provides less information to programmers in understanding the semantics supported by an infrastructure. The composers they write are therefore less automatic, relying more on users to make sense of the connections between devices in order to achieve their associated semantics. A generic interface that simply exposes a device's ability to exchange data, for example, does not provide programmers with specific information about the types and values of the exchangeable data and the kinds of devices with which the device can exchange data. A composer written for this interface must therefore rely on users making correct matches between devices. Thus, even with generic interfaces, a tradeoff still exists between ease of use and flexibility—the effort has simply been pushed to users. Below, we expand on these limitations using specific examples from ICrafter and Speakeasy.

Consider a composition of several lamps in a room under ICrafter's approach. Suppose that the lamps implement a `PowerSwitch` interface that declares a `power()` method for turning them on and off. A programmer can write a `PowerSwitchAll` composer for this programming interface that provides the ability to simultaneously turn the lamps on and off. Given the references to the lamps, this composer generates a user-interface for invoking the corresponding (power-all-lights) 'do all' operation. When the operation is invoked, the composer calls the well-known `power()` method of each lamp.

The `PowerSwitchAll` composer raises an important tradeoff that the programmer must make in writing composable programming interfaces. Which programming interfaces should a device implement? Two extreme options are a programming interface for all operations of a device or a programming interface for each operation. The first piece of code below is an example of the former approach while the second demonstrates the latter:

```

1)
    public interface Light {
        public void power();
        public void dim();
        public void brighten();
    }

2)
    public interface PowerSwitch {
        public void power();
    }
    public interface DimSwitch {
        public void dim();
    }
    public interface BrightenSwitch {
        public void brighten();
    }

```

In the first case, it is not possible to write a generic composer for devices implementing different programming interfaces even if they have common operations. For example, it is not possible to write a composer for simultaneously invoking the power operations of a TV and a light, since they provide different sets of operations. The latter approach overcomes the limitations of the earlier. However, it leads to a proliferation of programming interfaces and associated composers. Supporting the ‘Power All’, ‘Dim All’, and ‘Brighten All’ operations on the lights requires three separate programming interfaces and composers. An intermediate approach that defines programming interfaces for subsets of operations offers intermediate degrees of composition flexibility and programming cost of these two extreme approaches. As we will show later, our work does not force programmers to make this tradeoff between composition flexibility and automation.

Let us consider a different composition scenario, which involves a transfer of images in a camera to a display device for viewing. Assume that the camera and display device implement a `DataProducer` and `DataConsumer` interface respectively. The `DataProducer` interface declares a `produce()` operation that returns a value to transfer, and the `DataConsumer` interface declares a `consume()` operation that accepts the value. A programmer can now write a `DataPipe` composer that allows the camera and display device to exchange data. Given the references to the camera and display device, this composer generates a user-interface for invoking the corresponding (image) data transfer operation. Once the user invokes the operation, the composer calls the well-known

methods of its associated programming interfaces to achieve the image transfer. That is, it makes a call that passes the value returned from `camera.produce()` as an argument to `display.consume()`.

An issue with performing data transfers is how `DataProducer` and `DataConsumer` interfaces declare the data they exchange. Two options are to declare data as: (a) a generic object or (b) a programmer-defined type. In Java, the class `Object` demonstrates this notion of a generic data type. All classes in Java are subclasses of `Object` and can therefore be typecasted to it. Using generic objects, the two programming interfaces would be:

```
public interface DataProducer {
    public Object produce();
}
public interface DataConsumer {
    public void consume(Object x);
}
```

Here, the producer returns a value of type `Object` and the consumer accepts a value of that same type. Below is an example of using programmer-defined types in which the consumer and producer specifically exchange a `Picture` object:

```
public interface PictureProducer {
    public Picture produce();
}
public interface PictureConsumer {
    public void consume(Picture x);
}
```

These two options raise a subtle tradeoff a programmer must make between type flexibility and programming cost. The benefit of the generic approach is that it would require implementing only one composer, a truly generic `DataPipe` composer, to accomplish data transfers. One drawback is that all consumers are able to arbitrarily match with all producers because they all produce and consume the same generic type. When interacting with many devices, this approach could result in lists of many false-positives—that is, matches between devices that cannot exchange data. An example of a false positive is a match between a camera that only produces picture objects and an alarm clock that consumes time objects. Another drawback of using generic objects is that a device can only consume or produce one kind of data because forcing the generic

type does not allow overloading of the `consume()` and `produce()` methods in the programming interface declarations. Therefore, the camera could not independently produce URLs to both pictures and recorded video.

Supporting programmer-defined types in programming interfaces reduces the production of false positives because it allows consumers and producers to be matched by the types they exchange. It also allows overloading of the `consume()` and `produce()` methods so that devices can exchange more than one data type. However, it incurs the costs of writing many composers that are specific to the data types that devices can exchange. To illustrate, it requires writing separate `PicturePipe` and `VideoPipe` composers so that the camera can transfer two different kinds of data types. As we will show later, our approach allows the writing of a single data transfer composer that supports type matching.

Speakeasy, which specifically adopts the notion of generic programming interfaces, shows that it is possible to avoid the interface proliferation problem. This problem is associated with systems, such as ICrafter, that support automatic matching of producers and consumers. Speakeasy, on the other hand, takes a manual matching approach. It provides a user-interface in which users, themselves, select and appropriately connect the devices of matching programming interfaces. Thus, it relies on users to not make false positives. To assist users in a data transfer, devices must implement operations that return objects that indicate the values (including type descriptions) they can exchange. For example, a digital camera would implement an operation that returns an object indicating that it stores images as JPEGs. Also, a display device would implement an operation returning an object indicating that the device only displays GIF images. A user would discover that these two devices are incompatible for data transfer by comparing their supported picture formats.

The builders of Speakeasy performed user-studies to measure the burden of the manual connection approach on users. These studies show that for typical device users, this approach can be too low-level and difficult. Deferring the configuration task to ‘tech-savvy’ users is not always possible—especially during impromptu types of interactions.

In summary, ICrafter and Speakeasy offer the operation and data oriented frameworks necessary to meet our hypothesis. However, in order to offer high flexibility, they require a significant amount of effort from programmers or users. Our approach overcomes this problem by using programming patterns. Instead of requiring programmers to implement specific interfaces, it requires them to follow general device-independent patterns. Like the interface based approach of ICrafter and Speakeasy, programming patterns can be used to achieve an operation and data oriented framework. Our earlier description of how ObjectEditor generates command and state based GUIs from a single device's programming interface illustrates this ability (Section 2.1.6). Recall that under the ObjectEditor framework, method signatures following Java Bean conventions can be used to describe the state properties of an object. Non-bean patterns are also supported by ObjectEditor, however, they are not used in our work in composition. Signatures that are not used to export bean properties, therefore, describe operations. Figure 70 illustrates this process using a lamp's programming interface. In the next section, we describe how our infrastructure applies Bean and other patterns to overcome the various limitations just described and thus meet our hypothesis that it is possible to provide high-level and flexible composition support.

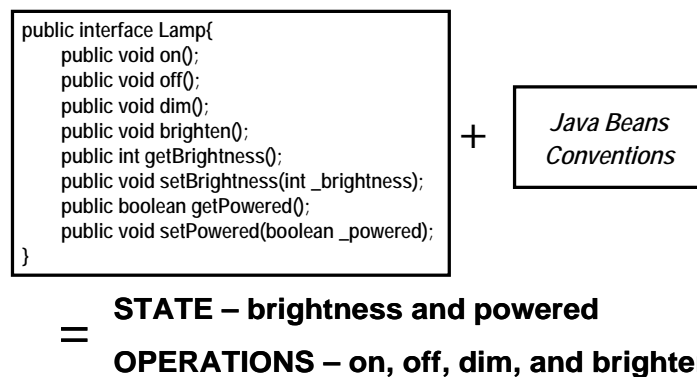


Figure 70. Extracting the state and operations from a lamp's programming interface.

6.2 Algorithms and Evaluation

We implemented prototype GUI-based composers based on the operations we identified. The prototypes were all written in Java and have been demonstrated to compose actual devices that were networked in the manner described in Section 3.1. Below, we describe

the algorithms supported by these composers using pseudocode. These descriptions include the programming patterns on which our composers rely.

6.2.1 'GUI Stack' Composer

This composer supports the 'GUI stack' operation, which stacks the individual user-interfaces of a set of devices into one. It implements the pseudocode below:

```
S = a set of device references  
guiStack(S) {  
    frame = createFrame();  
    for each device reference (x) in S {  
        p = ObjectEditor.generateCommandAndStatePanel(x);  
        frame.add(p);  
    }  
}
```

Our algorithm creates a GUI that displays the operations and state properties of a set of devices. Given the references to these devices in set *S*, it first creates an empty frame that will enclose the GUI. Then, for each device, it calls on ObjectEditor to create a panel displaying the operation and state of the device. This call, ObjectEditor.generateCommandAndStatePanel(), returns an appropriately filled panel, which the algorithm adds to the enclosing frame. Like Hodes' System, our system can stack panels horizontally or vertically. To support both, we actually implemented two composers based on the algorithm just described. The only difference between the two is that one composer implements a horizontal-based panel alignment while the other implements a vertical-based one. Figure 71 shows a movie watching GUI for a TV, DVD player, and receiver that is based on vertical stacking. All user actions on a device's panel, e.g. pushing a TV button, are handled by ObjectEditor as if they were performed on a single device GUI.

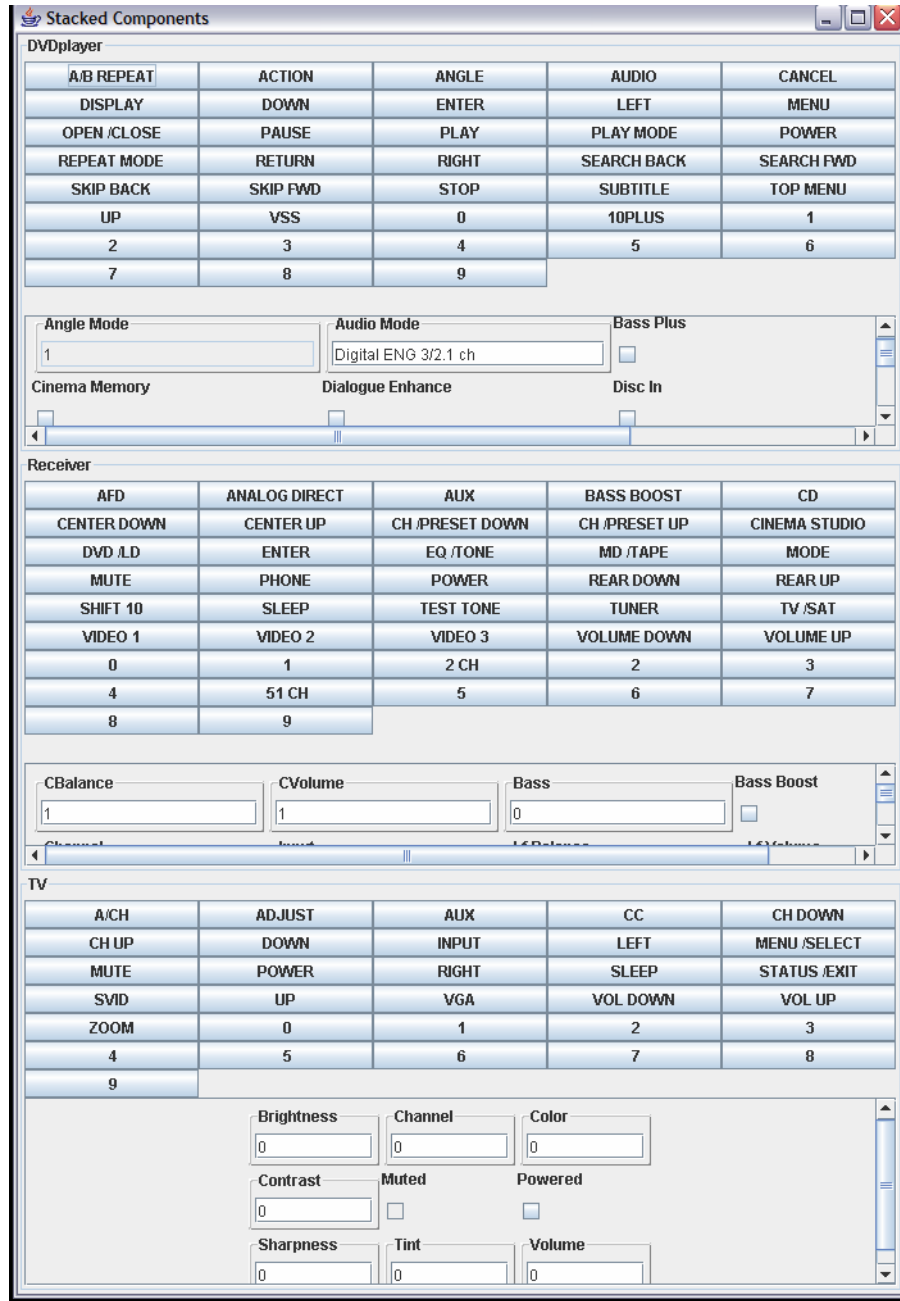


Figure 71. A stacked GUI for watching movies—based on the author’s TV, DVD player, and receiver.

6.2.2 ‘GUI Merge’ Composer

This composer supports the ‘GUI merge’ operation, which merges the operations of a set of devices to create a task-specific GUI. It implements the pseudocode below:

S = a set of device references

```
guiMerge( $S$ ) {  
    selectionFrame = createFrame();  
    for each device reference ( $x$ ) in  $S$  {  
         $p$  = generateOperationSelectionPanel( $x$ );  
        selectionFrame.add( $p$ );  
    }  
     $d$  = generateDoneButton();  
    selectionFrame.add( $d$ )  
}  
  
donePushed(selectionFrame,  $S$ ) {  
    mergedGUI = createFrame();  
    for each device reference ( $x$ ) in  $S$  {  
         $C$  = getChosenCommands(selectionFrame,  $x$ )  
         $f$  = generateFilteredPanel( $x$ ,  $C$ )  
        mergedGUI.add( $f$ );  
    }  
}
```

Given the references to several devices in set S , the composer first generates a GUI that allows a user to select the device operations that make up a target task (Figure 72). This process involves initially creating an empty frame. For each device, the composer calls `generateOperationSelectionPanel()` to generate a panel displaying the device's command names with a checkbox next to each name. It adds each panel into the frame. After adding all panels, the composer adds a done button to the bottom of the frame. At this point, a user can click the checkbox of each command that is desired in a target task. When finished, the user must simply click the done button. In return, the composer executes `donePushed()`, which extracts the selected commands and generates a merged GUI displaying them. The layout method of merged GUIs follows that of stacked GUIs—merged GUIs simply omit non-task specific buttons.

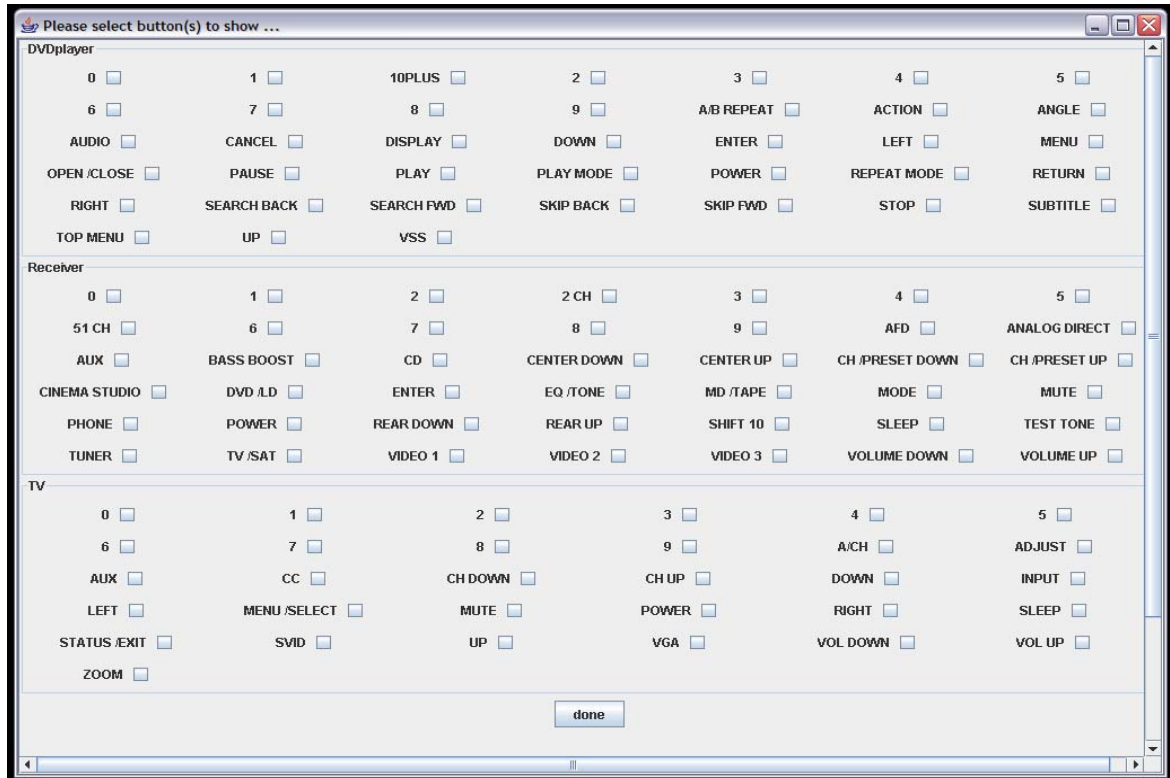


Figure 72. A GUI for selecting desired buttons for a target task.

6.2.3 'Do Sequence' Composer

This composer supports the 'do sequence' operation, which invokes a macro spanning several devices. It implements the pseudocode below:

S = a set of device references

```
doSequence( $S$ ) {
    frame = createFrame();
    for each device reference ( $x$ ) in  $S$  {
         $p$  = generateOperationSelectionPanel( $x$ );
        frame.add( $p$ );
    }
     $v$  = generateVerifyAndDonePanel();
    frame.add( $v$ );
    trackSelectionOrder();
}

donePushed() {
     $l$  = getButtonLabel();
     $b$  = new Button( $l$ );
     $C$  = getDeviceandOperationSelectionOrder();
    mapButtonToDeviceAndOperationOrder( $b, C$ );
}
```

pushed = a reference to the pushed button

```
doSequencePushed (pushed) {  
    C = getDeviceAndOperationSelectionOrder (pushed) ;  
    for each device reference and operation name pair (d,p) in C  
    {  
        [d,m] = getDeviceandActualMethod (d,p) ;  
        invoke (d,m) ;  
    }  
}
```

Given the references to several devices in set *S*, the composer first generates a GUI that allows a user to select the device operations that make up the desired ‘do sequence’. This process involves initially creating an empty frame. For each device, the composer then calls `generateOperationSelectionPanel()` to build a panel that lists all of the device’s operation names. Each panel also contains a checkbox next to each operation name. A user clicks these checkboxes to define the operations that make up a desired ‘do sequence’. The checkbox clicking order defines the order the composer invokes the associated operations. After all panels, the composer creates a user verification panel by invoking `generateVerifyAndDonePanel()`. This panel, which is added to the bottom of the frame, contains two textboxes and a ‘done’ button. One textbox displays the order of the sequence defined by the user, and the other allows the user to enter a string for labeling the resulting ‘do sequence’ button. The user can finalize a configuration by clicking the ‘done’. After building this GUI, the composer begins to track the order of device-operation pairs the user clicks using checkbox event listeners.

Figure 73 shows an example GUI for defining the ‘watch a DVD’ button mentioned earlier. Once a user clicks the ‘done’ button, the composer invokes `donePushed()`. This method builds the actual ‘do sequence’ button using the information provided by the user. Specifically, it creates the button *b* labeled after the user provided string and maps *b* to an ordered list, called *C*, consisting of the tracked sequence of device reference and operation name pairs. This mapping ensures that the appropriate invocations are made once the user pushes the button. The method `doSequencePushed()` handles such an event. It accepts a reference to the pushed button and retrieves the sequence of device reference and operation name pairs to which the button is mapped. For each pair in the

sequence, it invokes `getDeviceandActualMethod()` to extract the corresponding device reference and associated method object. It then invokes the method on the device.

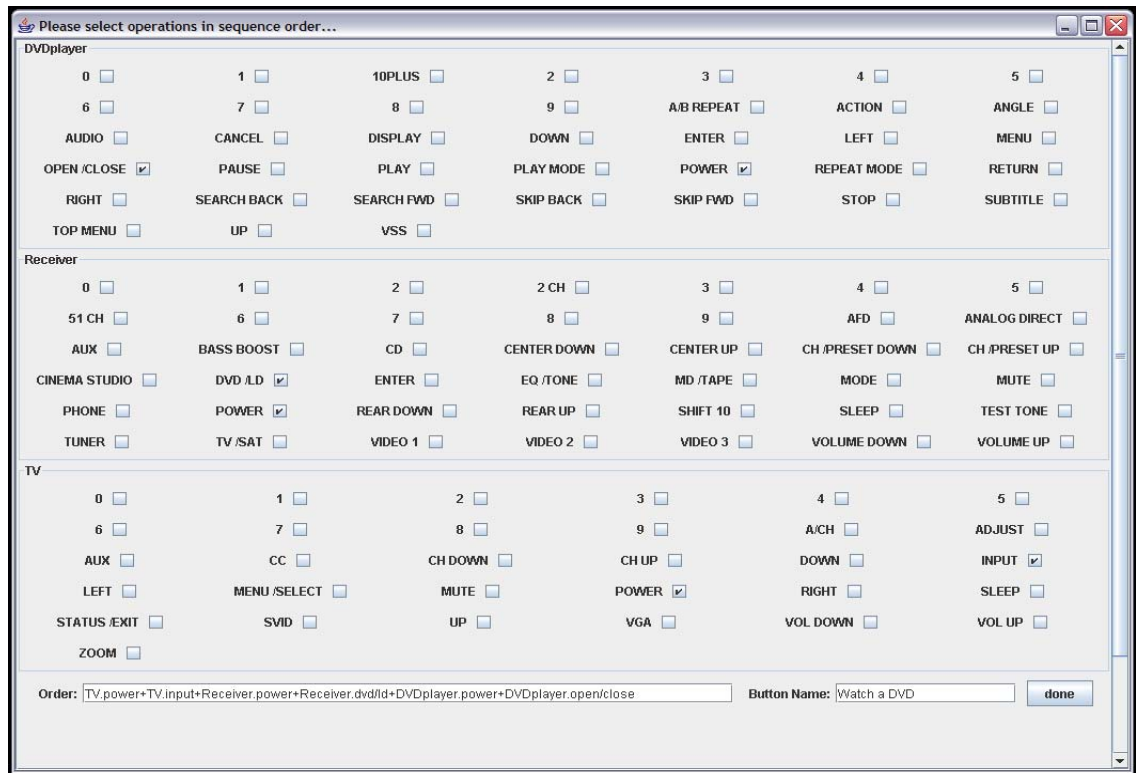


Figure 73. A GUI for creating a ‘watch a DVD’ button.

6.2.4 ‘Do All’ Composer

This composer supports the ‘do all’ operation, which invokes a command on each member of a set of devices that causes the devices to perform a similar action. It implements the pseudocode below:

S = a set of device references

```
doAll( $S$ ) {
    [ $O, D$ ] = getSharedSignaturesAndDevices( $S$ );
    frame = createFrame();
    for each operation signature ( $x$ ) in  $O$  {
         $b$  = generateButton( $x$ );
         $T$  =  $D.get(x)$ ;
        setButtonTargets( $b, T$ );
        frame.add( $b$ );
    }
}
```

b = a reference to a pushed button

```

doAllPushed(b) {
    T = getButtonTargets(b)
    for each device reference d in T
        m = getMethodFromButtonName(b.getLabel(), d);
        invoke(d,m);
    }
}

```

Given the references to several devices in set *S*, our ‘do all’ composer searches the devices’ programming interfaces to find the operation signatures that two or more of them share. This search, performed in `getSharedSignaturesAndDevices()`, returns two values: (1) the list *o* consisting of these shared signatures and (2) the hashtable *D*, which maps each signature to a list of references to devices offering the associated operation. For the set of room lamps in the earlier example, `getSharedOperationsAndDevices()` would return the signatures for the `power`, `dim`, `brighten` operations in *o* and the references to all the lamps in the room in *D* since the lamps share the same signatures. This composer and all others that we describe below extract their needed operation and state information from a device’s programming interface in the manner just described for `ObjectEditor`.

Given *o* and *D*, the composer then generates a GUI that displays all the ‘do all’ possibilities of the initial set of devices. It achieves this process by first creating an initially empty GUI frame. Then, for each signature in *o*, it creates an appropriately labeled button, maps the button to all the matching devices in *D*, and adds the button to the frame. Figure 74 shows the GUI it creates for a set of room lamps. Once a user pushes a ‘do all’ button (e.g. ‘dim All’), the composer invokes `buttonPushed()`. This method retrieves the references of devices that implement the associated operation (e.g. ‘`dim()`’) and invokes the operation on each device.

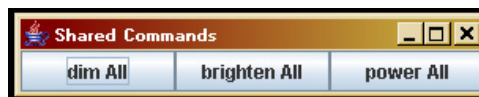


Figure 74. A ‘do all’ GUI for a set of lamps.

Our composer relies on device programmers to declare operations that perform similar actions with the same signatures. This convention allows the composer to automatically discover ‘do all’ operations for a set of devices, regardless of the interfaces implemented

by the devices. We believe that this approach is reasonable, specifically in the scope of devices, since many different brands and types of devices already follow conventions in the naming of their operations. This notion is illustrated by device control panels, remotes, and manuals.

6.2.5 Query Composer

This composer supports the query operation, which provides the ability to search a set of devices to find those with attributes of certain user-defined values. It implements the pseudocode below:

Pseudocode:

```

S = a set of device references

query(S) {
    frame = createFrame();
    P = getAllPropertyNames(S);
    for each name (x) in P {
        q = generateQueryEntryPanel(x);
        frame.add(q);
    }
    r = generateRunQueryButton();
    frame.add(r)
}

runQueryPushed(S) {
    F = getFilledQueryEntries();
    for each property name and expression pair [n,e] in F {
        S = S ∩ getMatches(S, [n,e]);
    }
    return S
}

```

Given the references to several devices in set *S*, the query composer supports queries by using the devices' properties as query attributes. The use of properties is appropriate because they represent a device's observable state and subsequently the fields that users can consider in a query. The composer first creates an empty frame. It then calls `getAllPropertyNames()` to build the set *P* containing the names of all properties of the available devices. Assuming that the composer is given the references to a set rainfall sensors with the programming interface below, this method would return `AverageRainFall`, `CurrentRainFall`, and `Location`.

```

public interface RainfallSensor{
    public void power();
    public void sleep(int x);
}

```

```

    public double getAverageRainFall();
    public double getCurrentRainFall();
    public String getLocation();
}

```

Recall from Section 2.2.1 that one of the problems of Cougar and TinyDB is that they require manually exporting device attributes to database relations in order to perform searches. By using Java Bean patterns, our composer is able to automatically extract these attributes.

For each property name in P , the composer invokes `generateQueryEntryPanel()` to generate a query entry panel and adds it to the frame. Each panel contains a textbox, labeled after a property name, that allows a user to enter a boolean expression describing a desired range of values for that property. After generating and adding the panels for all properties, the composer adds a ‘run query’ button. At this point, a user can begin defining a query by entering boolean expressions for the displayed properties. Figure 75 shows an example of our composer supporting a Cougar-like query of retrieving the references to all rainfall sensors in *Tompkin County*.

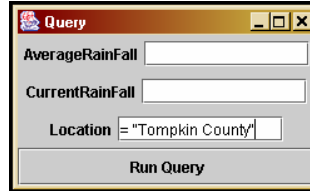


Figure 75. An example GUI for querying rainfall sensors with several attributes.

Once all entries are made in the query GUI, the user must click the ‘run query’ button. The composer then executes `runQueryPushed()`. This method invokes `getFilledQueryEntries()` to retrieve the list F , which contains of all the filled query entries from the GUI. Each element in F contains a pair $([n, e])$ consisting of a property name and a boolean expression that the user entered for the associated property. In our current example, F would only contain the pair $['Location', '=Tompkin County']$. To complete the query, the composer performs a loop that intersects the matching devices of each expression to the set of all devices. A device is a match of a given expression if the values returned by the getter methods of all properties that are referenced in the expression meet their corresponding queried values. After making all comparisons, the

composer then returns the references of remaining set of devices. For the rainfall sensor query, the composer would return the references of all sensors whose `getLocation()` method returns ‘Tompkin County’. To further interact with the matching sensors (e.g. view their average rainfall), the user can pass their references to any of the composers described above and others below (e.g. the UI merge composer).

6.2.6 ‘Data Transfer’ Composer

This composer supports the ‘data transfer’ operation, which allows information from a data producer to be transferred to a data consumer.

Pseudocode:

```

S = a set of device references

dataTransfer (S) {
    frame = createFrame();
    [ReadablePropertiesToProducers, ConsumersAndWriteableProperties]
    = searchForConsumersAndProducers (S);
    for each pair [c, W] in ConsumersAndWriteableProperties {
        consumerPanel = emptyConsumerPanel (c);
        for each property name and type pair (p, t) in W {
            transPanel = newTransferPanel (p,
                ReadablePropertiesToProducers.get (p, t));
            consumerPanel.add(transPanel);
        }
        frame.add(consumerPanel);
    }
}

producer = a reference to the selected producer
consumer = a reference to the selected consumer
property = the name of the property involved in the exchange

transferPushed (producer, consumer, property) {
    gm = getGetterMethod (producer, property);
    sm = getSetterMethod (consumer, property);
    value = invoke (producer, gm);
    invoke (consumer, sm, value);
}

```

Given the references to several devices in set *S*, our composer supports the data transfer operation by allowing devices to exchange property values. First, it generates an empty frame that will display the necessary components for achieving such transfers. It then executes the `searchForConsumersAndProducers()` method, which performs several

functions and returns multiple values. In particular, the method searches the programming interfaces of the devices to find all of the properties that one or more devices produce. A producer of a property implements a public operation for reading the property's value while a consumer implements a public operation for writing the value. Our implementation uses the Java Bean convention of describing readable and writeable properties. A property is readable if it has a public 'getter' operation, and it is writeable if it has a public 'setter' operation. As the below camera and display programming interfaces show, the camera and display device are `PictureURL` producers since they implement `getPictureURL()`.

```
public interface Camera{
    public void power();
    public void snap();
    public URL getPictureURL();
}
public interface Display{
    public void power();
    public void display();
    public void setPictureURL(URL x);
    public URL getPictureURL();
}
```

The display is a consumer since it implements `setPictureURL()`. The following alarm clock is a `Time` consumer and producer since it implements `setTime()` and `getTime()`, respectively.

```
public interface AlarmClock{
    public void power();
    public void snooze();
    public void alarmOff();
    public void setTime(Time x);
    public Time getTime();
    public void setAlarmTime(Time x);
    public Time getAlarmTime();
}
```

The atomic clock, given below, is also a `Time` producer since it implements `getTime()`. It independently sets its own time by accessing atomic clock radio signals—hence it does not offer a public `setTime()` operation.

```
public interface AtomicClock{
    public void power();
    public Time getTime();
}
```

The method `searchForConsumersAndProducers()` creates and returns a hashtable called *ReadablePropertiesToProducers* that maps each readable property's name and type pair, from the entire set of devices, to its corresponding list of producers. In addition, the method creates and returns a list of pairs called *ConsumersAndWriteableProperties*, with each pair containing: (1) a device reference and (2) a list of each writeable property name and type pair of the device. The composer performs all of these tasks within this single method in order to avoid repeating device programming interface searches. The method would, for example, return the following values if given a set of references to two cameras and two display devices with the programming interfaces defined above:

Mappings of *ReadablePropertiesToProducers*:

```
[(PictureURL, URL) → (Camera1Ref, Camera2Ref, Display1Ref,
Display2Ref)]
```

Elements of *ConsumersAndWriteableProperties*:

1. (Display1Ref, [(PictureURL, URL)])
2. (Display2Ref, [(PictureURL, URL)])

After completing the method, the composer begins adding to the empty frame. For each pair in *ConsumersAndWriteableProperties*, the composer creates a new panel (*consumerPanel*) and fills it with the necessary components to display the data transfer possibilities for the pair. This process involves invoking `newTransferPanel()` to create a panel (*transPanel*) for each writeable property name and type pair of the given consumer. Each panel contains a drop-down-box that allows a user to select the producer device from which to transfer values onto the panel's associated writeable property. The method retrieves this list of producers from *ReadablePropertiesToProducers* by hashing a given writeable property name and type pair. Our composer, thus, relies on device programmers to name properties of similar semantics and types with the same name. Following this convention insures type correct connections between arbitrary consumers and producers within our composer. Further, it allows the composer to maximize the number of meaningful connections and minimize the problem of false positives connections.

To allow users to activate a selected configuration, each *transPanel* contains a 'Transfer' button. The composer adds each *transPanel* created for a given consumer to

the consumer's corresponding *consumerPanel*. Figure 76 shows the data transfer GUI created for our example set of cameras and display devices.

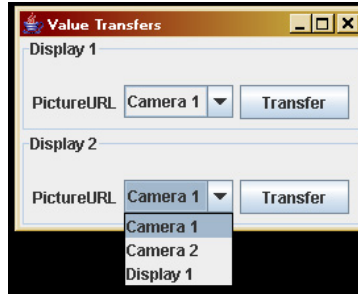


Figure 76. A data transfer GUI for cameras and display devices.

Once a user clicks a ‘Transfer’ button, the composer invokes `transferPushed()`. This method accepts the name of the property involved in the transfer and references to the selected producer and consumer. It gets the producer’s getter method for the property and then invokes the method. This call returns the producer’s value of the property. The composer then passes this value to the consumer by invoking its setter method.

6.2.6 ‘Conditional Connect’ Composer

This composer supports the ‘conditional connect’ operation, which automatically invokes one or more operations on a set of devices based on the state conditions of another set of devices. It implements the pseudocode below:

S = a set of device references

```
conditionalConnect( $S$ ) {
    frame = createFrame();
    conditionsPanel = newConditionsPanel();
    eventsPanel = newEventsPanel();
    for each device reference ( $x$ ) in  $S$  {
        statePanel = generateStateEntryPanel( $x$ );
        conditionsPanel.add(statePanel)
        operationPanel = generateOperationCheckPanel( $x$ );
        eventsPanel.add(operationPanel);
    }
    addConnectButton();
}

connectPushed(conditionsPanel, eventsPanel) {
     $E$  = getEnteredEvents(eventsPanel);
     $C$  = getEnteredConditions(conditionsPanel);
    for each device reference, property name, and expression
    triple
```

```

        ([d,p,e]) in C
        monitorPropertyChanges(d);
    }

```

C = the list of filled conditions, where a condition is a triple containing a device reference, property name, and expression $([d,p,e])$

E = the list of filled events, where an event consists of a device reference and operation name pair $([d,o])$

```

propertyChanged(d,p,E,C) {
    if allConditionsMet(C)
        invokeAllEvents(E);
}

```

Our composer allows a user to enter conditions for the properties of a set of devices that should trigger a set of operations (or events) on another set of devices. Given the references to several devices in set S , it first creates an empty frame that it will build to accept this user input. It divides this frame into two panels—one for entering conditions and another for selecting matching events. For each available device, the composer creates: (a) a state-based panel for entering the conditions for the device’s properties and (b) an operation-based panel for selecting potential events. It creates these panels by invoking `generateStateEntryPanel()` and `generateOperationCheckPanel()`, respectively. The state-based panel lists all property names of each device and a matching textbox for each name. Users enter their desired conditions for a given device’s property in its corresponding textbox. The operation-based panel lists all of the operation names of each device and a matching checkbox for each name. Users select a desired event by clicking on the corresponding operation’s checkbox.

After adding each device’s state and operation based panel, the composer invokes `addConnectButton()` to add a ‘connect’ button for users to click to activate their configurations. Figure 77 shows an example GUI for achieving the earlier mentioned adhoc security system involving a motion sensor and some lamps.

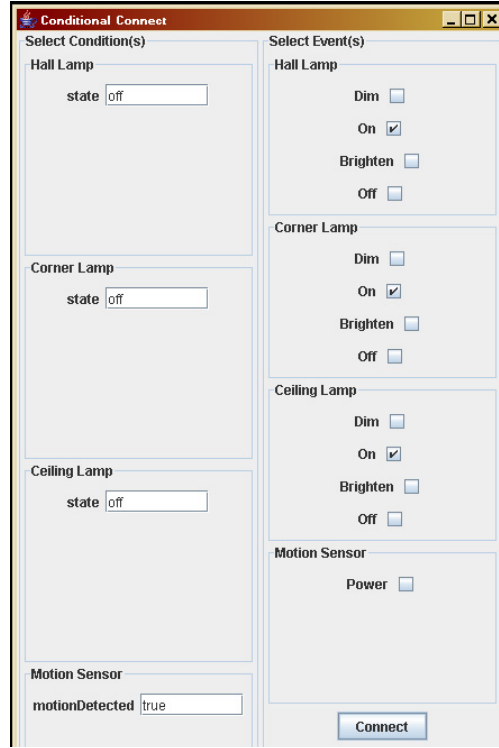


Figure 77. A 'conditional connect' GUI for creating the adhoc lamps and motion detector security system.

Once a user clicks the connect button, the composer invokes `connectPushed()`. This method extracts all of the entered conditions and events from the GUI. It creates two lists, C and E , which encapsulate this information. List C contains the filled conditions, where a condition is a triple containing a device reference, property name, and user-entered expression $([d, p, e])$. List E contains the filled events, where an event consists of a device reference and operation name pair $([d, o])$. For each device in C , the composer begins to monitor its state changes. It uses the notification mechanism proposed by Java Beans to monitor device property changes. That is, each device (object) informs a list of listener objects of its property change events. The 'conditional connect' composer thus registers itself as a listener of all devices that make up the set of user-specified conditions. For each change notification, the composer invokes `propertyChanged()`, which accepts E , C , a reference to the corresponding device, and the name of the changed property. This method checks to see whether the change is sufficient enough to meet all conditions specified in C . If so, it invokes all events listed in E .

6.3 Conclusion

This chapter presents the idea of *pattern-based composition*. It abstracts several new and existing composition semantics into a set of abstract operations and shows that existing infrastructures cannot support all of them at a high-level. Through the use of programming patterns, we were able to build a composer for each identified operation. As presented, each composer provides a user with a user-interface for easily performing its corresponding operation. The implementation thus proves our *High-level and Flexible Composition Hypothesis*: a new infrastructure can be built that supports the composition semantics of existing high-level infrastructures and provides higher-level support than all other infrastructures that can support all of these semantics.

Being based on programming patterns, our approach relies highly on programmers to follow certain conventions when coding devices. However, we do not consider this reliance as a limitation since following an interface, as required by other composition infrastructures (ICrafter and Speakeasy), is also a special kind of convention. More important, following programming patterns allows for program understandability. In fact, some infrastructures systems such as UPnP insist on common conventions for device operations.

Chapter 7: User-Based Composition

As described in the previous chapter, programming patterns allow us to write mechanisms for achieving high-level support for several composition semantics. Our ‘do all’ composer, for example, *automatically* discovers all possible ‘do all’ operations of a set of devices and creates a user-interface for invoking the operations. Similarly, our data transfer composer *automatically* discovers all type-correct data transfer possibilities of a set of devices and generates a user-interface for performing the exchanges. Even with such mechanisms, our pattern-based framework still has some important limitations.

Consider the ability to create ‘do sequence’ operations that invoke certain sequences of commands on devices. The previous chapter shows an example of such an operation invoking the following commands on a TV, DVD player, and receiver:

- 1) Turn on the TV
- 2) Set TV to DVD video input channel
- 3) Turn on the receiver
- 4) Set the receiver to DVD audio input
- 5) Turn on the DVD player
- 6) Open the DVD player’s disc tray

The operation prepares the devices for movie watching to a point that a user must simply place the desired DVD in the disc tray and press play. After watching the DVD, the user might wish to watch TV and thus invoke a ‘do sequence’ operation that performs the following:

- 1) Set TV to cable box input channel
- 2) Set the receiver to cable box audio input
- 3) Turn on the cable box

It is likely that people will have many kinds of multi-device tasks in which the efficiency offered by ‘do sequence’ commands is highly desirable (e.g. listening to music and playing video games). High-level ‘do sequence’ discovery in an infrastructure is thus appealing. However, current discovery approaches, as supported by our ‘do sequence’ composer and Palm/Pocket PC programs, require users to manually define ‘do sequence’

operations themselves. The process involves users selecting the commands and defining the sequence order of an operation. This approach is unlike our pattern-based ‘do all’ algorithm, which automatically discovers useful ‘do all’ operations for arbitrary devices. It can become tedious as users increasingly want ‘do sequence’ operations for efficiently performing their device interactions. Further, it is open to human error. The user-specific nature of ‘do sequence’ operations, however, prevents patterns from being used to automatically discover useful ‘do sequences’ for arbitrary devices like for ‘do alls’. Though patterns can be used to expose composability, it is not logical to define ‘do sequence’-based patterns for every possible user’s behavior and device arrangement (e.g. specific devices in a home theater).

Our pattern-based framework exhibits a similar limitation in its support of the conditional connect operation. Like the definition of ‘do sequence’ operations, the definition of conditions and matching events is user influenced. Therefore, these components of a ‘conditional connection’ cannot be automatically discovered using patterns. Our conditional connect composer thus currently implements the approach of using manually provided definitions. That is, it requires a user to fully define the components of a conditional connection using a user-interface displaying them (Figure 77).

Yet another related limitation of our pattern-based framework arises from the ‘GUI merge’ composer. Similar to the definition of ‘do sequence’ operations and ‘conditional connections’, the definition of task-based user-interfaces is highly user influenced. Patterns cannot be used to expose the tasks of every possible user. The current approach of requiring users to explicitly define the set of commands for each task that they perform can be cumbersome and open to human-error.

In summary, the ‘do sequence’, ‘GUI merge’, and ‘conditional connect’ operations are highly based on the behaviors of specific users. The current approach of relying on users to make necessary device connections to support these operations can be tedious. Programming patterns, on the other hand, are ineffective in providing automatic connections like in the ‘do all’ and ‘data transfer’ composers. To offer higher-level

support, it becomes attractive to use machine learning (ML) to automatically discover user influenced connections from logs of users' interactions. The approach could, for example, allow our 'GUI merge' composer to automatically discover the commands of tasks that a specific user performs. Similarly, the 'do sequence' composer could automatically discover sequences of commands a user typically invokes and then create appropriate operations for invoking them. We found, however, that neither the manual or fully-automatic (ML-based) approach is optimal for discovering user influenced connections within composers. Each approach possesses certain advantages over the other.

In the following section, we describe our ML-based approach in more detail. We then discuss a set of experiments we performed to compare the approach with the manual approach. Based on these experiments, we present results that prove our hypothesis by showing the respective benefits of the two approaches. Finally, we present our conclusions.

7.1 ML Approach

The goal was to design an approach for automatically extracting groups of commands that are commonly used together from a given user's command history. Appropriately, each group would correlate to commands needed in a task or a 'do sequence' operation for that user. The approach does not currently address ML-based discovery of 'conditional connections', which requires the ability to access both the operations that users invoke and state of devices over time. Recall, however, that our users' interaction logs only consist of invoked commands since traditional remotes do not communicate state information.

Our goal requires a formal definition of what it means for commands to be related or commonly used together. In our approach, two or more commands are related if they predict similar behavior. DVD navigation buttons (up, down, left, right, enter) are related since each button tends to predict, for example, that another navigation button is likely to be invoked next, but the VCR's play button is not. A command can thus be associated with a probability distribution (or histogram) of commands that follow it. In other

words, each command associates to vector C_x . Each vector element, C_{xy} is a count of how many times command x follows command y . Thus, given a user's log of command accesses, it is possible to create a distribution for each command found in the log and compile these distributions into a matrix representing the whole data.

Following standard methods in information retrieval and work modeling human-human interactions[18, 30, 32], our approach computes the similarity of every command pair in the matrix by computing the cosine of the angle between the pair's associated vectors. This process is the same as computing the inner products of the vectors. The result is a similarity matrix that describes the similarity of a given command to every other command. Next, to find groups of related commands in the matrix, our approach uses the standard k -means algorithm[21]. Basically, the algorithm works by viewing the similarity matrix as a collection of points in space and divides the points into k clusters, where k is an integer-based input parameter. It begins by randomly selecting k centroids in the space defined by the points. Then, it designates each point to the nearest centroid. Next, the algorithm moves each centroid so that it is the mean of the data points assigned to it. The algorithm then reassigns points to their nearest centroid. It repeats the process of moving centroids and designating points until the centroids no longer change. When the algorithm terminates, it returns a set of command clusters that are as well divided as possible.

7.2 Experiments

In order to compare the manual and ML-based approaches, the users needed to actually complete some form of the manual approach. We simply used participants 1-10 in our logging study—i.e. everyone except the author. At the end of each logging period, we asked its participant to create paper-based user-interfaces consisting of buttons that were sufficient for his/her commonly performed tasks and desired 'do sequence' commands. The paper-based survey allowed us to evaluate the quality of user-interfaces the participants could create without requiring them to learn how to use special purpose software.

Specifically, each participant received a stack of several 6" x 11" sheets of white cardboard paper and a set of small squares displaying various remote control button names and icons (Figure 78). These squares were stuck on several sheets of paper using reusable putty, allowing users to easily move them between sheets. We asked the participants to imagine designing a remote containing groups of buttons that they commonly use together. We then showed them an Ipaq and told them to imagine that such a device could display the button groups using as many screens (or user-interfaces) as they wished. To create these user-interfaces, users simply looked over their actual remote controls for buttons required to complete their tasks, found the equivalent square button they wanted from the pool of squares, and stuck them on the appropriate cardboard sheet. Figure 79 shows the user-interfaces created by participant 5.

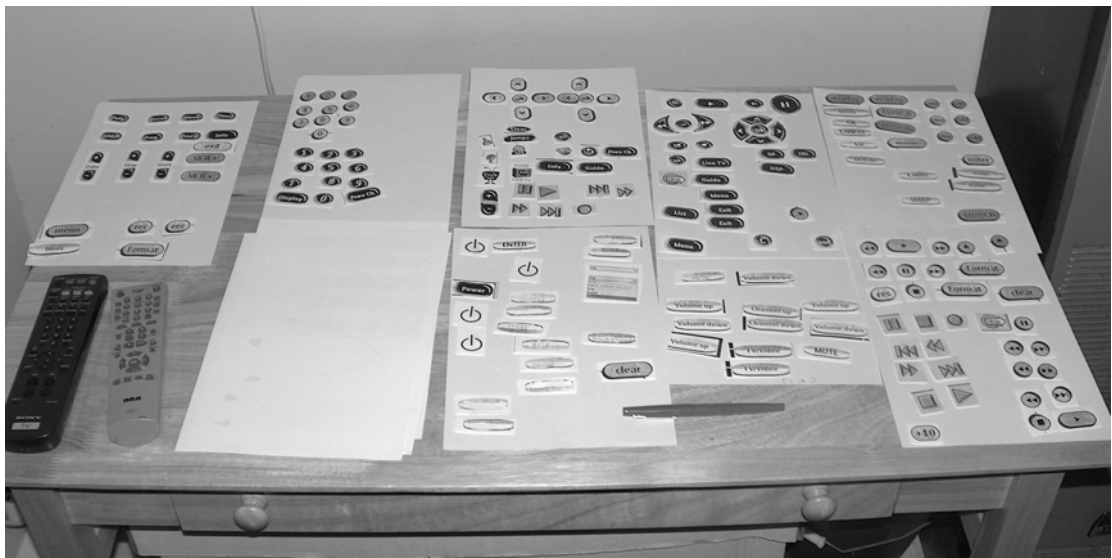


Figure 78. Our setup containing the participants' remote, several blank sheets (screens), and button squares.

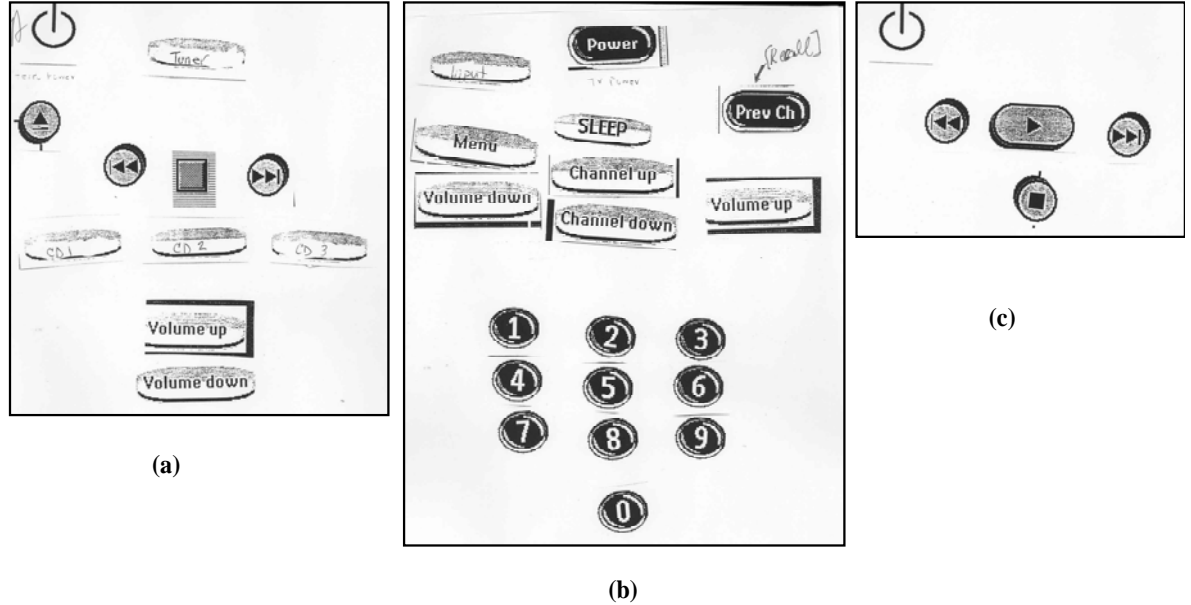


Figure 79. The three user-interfaces that P5 created.

After the participants created their initial user-interfaces, we told them to imagine having new buttons that could invoke sequences of commands that they commonly executed. Their task was to think of such sequences and create the associated buttons on blank squares. They simply labeled the square and stuck it on the screen where they wished to display it. We recorded the sequence of commands that users associated with these new ‘do sequence’ buttons. After collecting all the participants’ user-interfaces, we applied our ML-based approach on their respective logs to produce command clusters. The process involved varying the values of k for the k -means algorithm in order to find meaningful clusters.

7.3 Evaluation

We evaluated the performance of the ML and manual approaches using three criteria:

- 1) *Completeness* – How well does an approach produce task-based user-interfaces that contain all commands that a user needs?
- 2) *Task-based grouping* – How much does an approach require users to switch between tasks-based user-interfaces during a single task?

- 3) *‘Do Sequence’ discovery* – How well does an approach produce ‘do sequences’ that users actually need?

7.3.1 Completeness

Each approach has different limitations in its ability to yield complete user-interfaces. Consider the manual approach. Seven of the ten participants produced user-interfaces with at least one missing command. The number of omitted commands varied between zero and twelve, with no commonly omitted commands between users (Table 16).

Participant	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
Number of missed buttons	12	1	5	2	2	5	0	5	0	0

Table 16. A count of each participants missed buttons.

To illustrate the types of commands missed, let us consider P3. This participant missed her cable box’s page-up and page-down commands for browsing through channel listings, even though she used these commands 164 and 259 times respectively. In addition, she missed the TV’s channel up and down buttons and the cable box’s ‘c’ command, which she uses to set show reminders. In general, the results show that users are prone to creating incomplete user-interfaces—they even omit frequently used buttons.

The ML approach possesses a different kind of limitation. Because it employs a user’s actual interaction history, it has the disadvantage that it can only include commands found in the history. Consider P5, who designed a user-interface containing the CD1, CD2, and C3 commands of her stereo (Figure 79a)—these buttons play the CD in a given slot (1-3) in the stereo. During her logging period, she only played the CD in slot 1 by pushing CD1, thus, the ML algorithm did not include the CD2 and CD3 commands. P5 did mention that all CD slots in the stereo contained a CD throughout this period, but she only wanted to listen to the CD in slot 1.

7.3.2 Task-based Grouping

As mentioned in Section 5.1, we interviewed the participants to gather the tasks that each of them most commonly performed. The interviews show that each of them performed at least one task involving multiple devices. However, only half of the participants created user-interfaces consisting of commands from multiple devices. The other half created single-device user-interfaces, even though they were specifically told to create task-based user-interfaces. Figure 79 demonstrates this behavior: P5 created individual user-interface for her stereo (Figure 79a), TV (Figure 79b), and DVD player (Figure 79c) instead of creating a set of user-interfaces that spanned those devices. To quantify how well the participants created task-based groupings, we counted the number of screen switches for some common tasks. Table 17 compares these values with the number of single-device remote control switches involved in the same tasks. The data shows that only two out of ten participants created a set of user-interfaces that do not require switching during common tasks. The other users designed user-interfaces that require switching multiple times for a given task. P2 even created a set of user-interfaces that requires more screen switches than remote control switches while watching a DVD.

Under the ML approach, one and two weeks worth of logging users produces a mix of both appropriate and inappropriate clusters. Figure 80 provides an example. It shows a two-dimensional projection of P6's clusters. On the left side, cluster *a* contains several of the TV channel commands, which P6 uses together when watching TV. Cluster *b* contains the commands for preparing the VCR to record a future TV show, and cluster *c* contains the commands for watching VHS tapes on the VCR/DVD combo device. Each cluster, however, has commands that do not belong in the tasks to which they associate. For example, cluster *a* has the VCR/DVD combo's '8' command, which does not belong in a cluster for watching TV. Also, cluster *c* contains the VCR/DVD '7' command, which does not belong in the cluster for watching a VHS tape. Such cases of misplaced commands can cause users to switch between many different task-based user-interfaces during a task just to find a command. In the experiments, there were no users whose data yielded an entire set of clusters in which all commands within each cluster share a single task.

Participant	Task	# of devices required	# of switches	
			Single Remotes	User-created UI
P1	Watch a VHS tape	2	4	4
P2	Watch a DVD	3	5	3
P3	Watch cable TV (includes using the cable box's show listings to find a single interesting show to watch)	2	4	5
P4	Watch a DVD	2	4	4
P5	Watch a DVD	2	4	4
P6	Set up the VCR to record a future TV show	2	2	2
	Watch a VHS tape	2	2	3
P7	Watch cable TV (using TiVo)	2	4	1
	Use XBOX (to listen to MP3 music files)	3	4	1
P8	Watch a DVD	2	4	2
P9	Watch a DVD	2	4	2
P10	Watch cable TV	2	4	1

Table 17. A count of user-interface switches required for participants' common tasks.

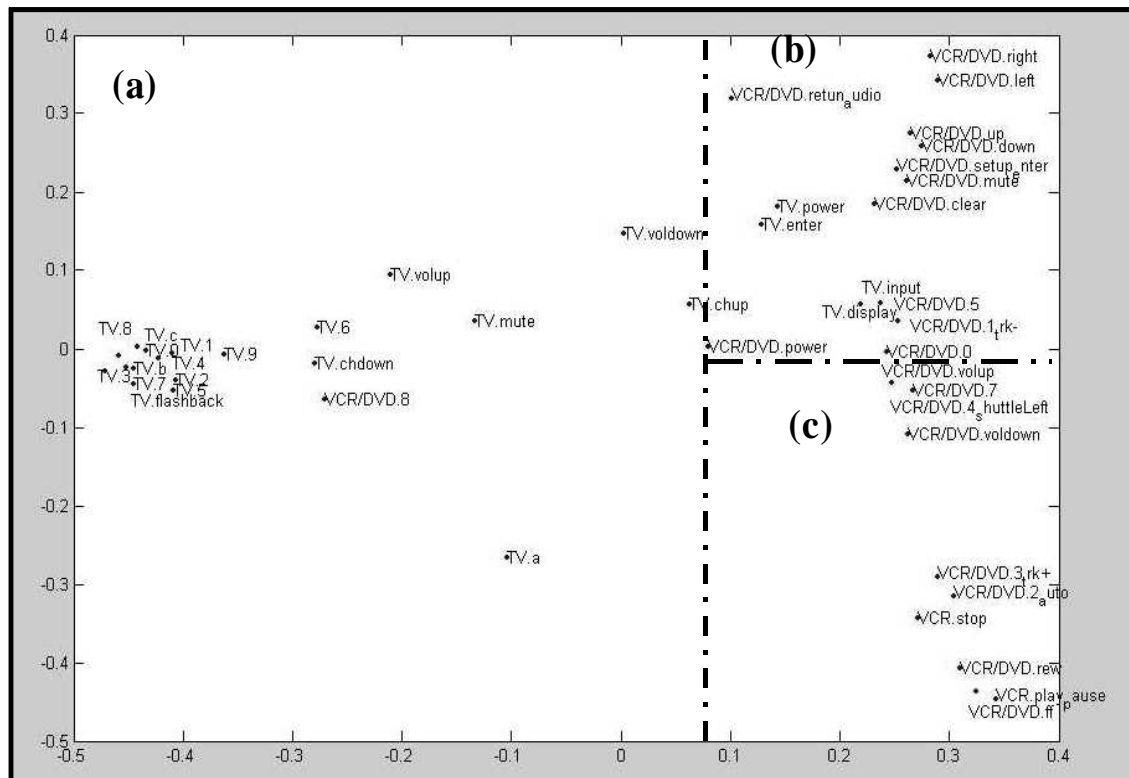


Figure 80. A projection of P6's clusters

The above example also illustrates another limitation of the k -means clustering algorithm used by our approach. It does not support multiple instances of a given command, thus it puts the command in only the cluster it believes is best fit. To illustrate, the algorithm places the TV power command in cluster b . However, all tasks require the ability to power the TV. Combining the clustering algorithm with heuristics could address this problem, as well as possibly preventing some cases of command misplacement. One such heuristic could be to always keep a device's number commands (e.g. TV channel number buttons and security system keypad numbers) in the same cluster, potentially the cluster that contains most of the number command. Alternatively, we could explore using another kind of clustering algorithm that does not associate a command to a single cluster.

In summary, the clustering algorithm supported by our approach is not optimal with a one or two week long user log. It is possible that the algorithm's performance would increase with more data. However, it is not likely that users would be willing to wait the time needed to produce such data before the algorithm generates results.

7.3.3 'Do Sequence' Discovery

In our experiments, all users requested at least one 'do sequence' operation. Most of the operations they requested were actually sequences of device commands that they commonly issued (as the logs show). However, two participants (P2 and P3) requested 'do sequence' operations that do not reflect sequences of commands they actually used. To illustrate, P2 requested two macros for (1) turning on the DVD changer and receiver in order to watch a movie and (2) turning on the DVD changer and receiver and then having the DVD changer play a music CD inside one of its 5 slots. P2 performed the particular tasks associated with the two macros several times over the week. However, the two sequences did not occur in his entire log. In fact, there was no instance of him ever using the receiver's power command during his logging period. When asked why he included the unused command, the participant admitted that he always left the receiver on. As a result, invoking any of the two requested macros would actually sidetrack him from performing the desired task because they would turn off the receiver.

Our ML approach, on the other hand, can only place commands in clusters based on how much the commands are used together. Because the commands that make up a ‘do sequence’ command are inherently part of the same task and a task can require other non-‘do sequence’ commands, a cluster for a given task can consist of both ‘do sequence’ and non-‘do sequence’ commands. Our ML approach cannot differentiate between the two kinds of commands and therefore it cannot identify macros on its own.

7.4 Conclusion

This chapter shows how the ‘do sequence’, ‘GUI merge’, and ‘conditional connect’ operations are highly based on the behavior of specific users. It describes how the current approach of relying on users to make necessary device connections, in supporting these operations, can be tedious. Programming patterns, on the other hand, are ineffective in providing higher-level support. Towards higher-level support, we present an ML-based approach for automatically discovering groups of commands that are used together. Appropriately, such groups could map to commands needed in a ‘do sequence’ operation or merged GUI.

From evaluating the ML and manual approaches, we found that neither approach is completely better than the other in defining ‘do sequence’ operations and tasks (for merged GUIs). In particular, our results show that users have inaccurate models of their own behavior, and the ML approach requires lengthy observation periods to discover accurate models. The next chapter, which describes the thesis’ overall conclusions and future work, describes our ideas for a better solution to defining ‘do sequence’ operations and tasks.

Chapter 8: Conclusions and Future Work

In addressing the user-interface deployment issue, this thesis makes several contributions:

- It provides a set of reasons for interacting with devices using software-based user-interfaces on mobile computers.
- It abstracts current forms of user-interface deployment into a set of high-level approaches and systematically evaluates them. Using a mix of quantitative and qualitative metrics, the evaluation shows the advantages and disadvantages of each approach. In particular, it verifies our *Uniqueness Hypothesis* that each approach offers a unique benefit, thus providing a reason why it exists.
- The thesis investigates, in depth, the promising user-interface generation approach which has the important limitation of taking a long time to create a user-interface. It shows that by retargeting user-interface, a generator can successfully overcome this problem. In particular, the thesis verifies our *Time-Efficient Generation Hypothesis* that it is possible for SUI and GUI generators to offer deployment times that are often as good as or noticeably better than the inherently fastest approach of locally loading device-specific user-interface code. It identifies various levels of retargeting a system can support and presents a set of algorithms that we used to achieve higher retargeting flexibility than previously supported. These algorithms have two important ideas—regression-based prediction and cache-based retargeting. Regression-based prediction allows a generator to use estimation functions to predict: (a) the fastest source user-interface to retarget and (b) whether to retarget or generate a new user-interface. Cache-based retargeting allows a generator to avoid the time cost involved in executing the prediction functions during an interaction time by using cached results from previous times.

- To further address the *Time-Efficient Generation Hypothesis*, the thesis presents the idea of history-based generation. It describes a straightforward approach to generating history-based user-interfaces and shows that it can be used to reduce SUI generation times down to client-factory like times. For GUIs, the approach is not as competitive. Although history-based GUI generation times are significantly lower than full GUI times, they are not as low as client-factory based times.
- The thesis also verifies the *Screen-Space-Efficient Generation Hypothesis* that history-based generation can additionally be used to create user-interfaces that consume significantly fewer screens than their corresponding full device user-interfaces. On the space constrained Ipaq Pocket PC, the full user-interfaces of several networked devices require two to three screens while their history-based user-interfaces only require one.
- The thesis also addresses the issue of how devices are composed. It identifies new and existing composition semantics that apply to a wide variety of devices and abstracts them into a set of operations. It summarizes existing composition infrastructures and shows how they are limited in simultaneously offering high-level and flexible support for the identified operations. The thesis verifies the *High-level and Flexible Composition Hypothesis* that it is possible to overcome the above limitation and build a new infrastructure that: (a) supports the composition semantics of existing high-level infrastructures and (b) provides higher-level support than all other infrastructures that can support all semantics. The idea is programming patterns. Programming patterns address the programming effort involved in supporting composition while ML addresses user effort.

Future work exists in further investigating our several hypotheses:

- *Uniqueness Hypothesis*: As mentioned earlier, we evaluated current user-interface deployment approaches using a mix of qualitative and quantitative metrics. Our qualitative comparisons show whether an approach is better than other under a

given metric, however, they do not show by how much. It is important to collect more quantitative results, particularly for the currently qualitatively-based metrics in our evaluation (e.g. maintenance costs).

- *Time-Efficient Generation Hypothesis:* Our current retargeting implementation only supports primitive-typed property widgets. It would be useful to extend our implementation to also support widgets that display structured types. This feature would, for example, allow our generator to retarget a widget showing a VCR's program record list to a TiVo's program record list and vice versa. Besides structured types, heterogeneous SUI retargeting could also be supported to see whether it offers any benefits as seen in the GUI case. This process requires profiling our SUI generator in order to discover the necessary regression-based prediction functions. The resulting profiling data could allow us to gain a better understanding of why there is such a large difference between GUI and SUI deployment times. It could also allow us to answer why the difference between SUI-based client-factory and generation times are relatively close in comparison to the difference between GUI-based client-factory and generation times.
- Future work also exists in history-based generation. It seems particularly attractive to combine the history-based generation and retargeting approaches. Alone, the approaches can offer generation times that are lower than client-factory based times. It may be that retargeting history-based user-interfaces can offer even lower times and potentially minimize the importance of certain speed enhancements such as cache-based retargeting. Collecting more real-world data from different users would allow for a deeper evaluation and further verification of our current results.
- *Screen-Space-Efficient Generation Hypothesis:* New user data could also be used to further verify the screen space efficiency measurements of our history-based generator. Our current data set, however, is mainly based on users' histories with entertainment devices (e.g. TVs, VCRs, and receivers). It would be useful to include new data involving interactions with a more diverse set of devices such as

projectors and thermostats. Further work is also needed to evaluate the benefits of history-based generation on cell phones, which have screens that are fractions of the size of the Ipaq's screen.

- *High-level and Flexible Composition Hypothesis* – Future interaction data could also motivate new useful composition semantics. With such examples, it is important to test the flexibility of our composition framework to see whether it can support them.

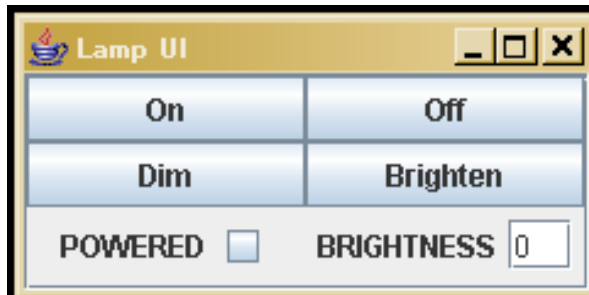
Since neither manual nor ML approach is completely better than the other in defining 'do sequence' operations and tasks (for merged GUIs). We believe that a better solution is to combine both approaches to yield mixed-initiative composers [17] that use the advantages of one approach to alleviate the disadvantages of the other. This solution could provide better support for:

- *Completeness*: Since users occasionally miss buttons, our ML algorithm could help provide completeness by validating the user-interfaces they create. Alternately, users could help the ML approach by providing it with a list of commands whose use it has not observed.
- *Task-based grouping*: One way to change the single-device-centric approach of users is to give them an initial set of task-based clusters provided by the ML approach. Users can then use their intuition to refine these clusters, given that the ML approach sometimes places commands in wrong clusters.

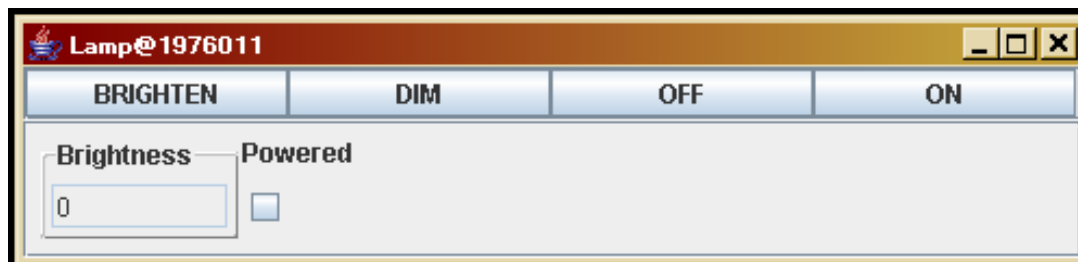
Users can also design their own task-based user-interfaces without any initially assistance from ML. Although it is likely that these user-interfaces will be single-device, rather than task based, they should at least reduce the number of displayed buttons. A ML algorithm could then migrate users toward a more optimal solution by observing users' interactions with their designed user-interfaces and periodically offering suggestions for new designs when it has observed enough data to be confident in an improvement.

Appendix A: Snapshots of Predefined and Generated GUIs

1) The predefined lamp GUI:



2) The lamp GUI generated by ObjectEditor:



3) The predefined TV GUI:

The TV UI window features a standard Windows-style title bar with a red icon, the text 'TV UI', and minimize, maximize, and close buttons. The main interface is organized as follows:

- Top Bar:** Contains 'ZOOM' and 'STATUS/EXIT' buttons.
- Navigation Grid:**

	UP	
LEFT	MENU/SELECT	RIGHT
	DOWN	
- Volume and Channel Controls:**

+	A/CH	+
VOL		CH
-	MUTE	-
- Numeric Keypad:**

ADJUST	1	2	3
CC	4	5	6
	7	8	9
		0	
- Input Selection:** A row of buttons: POWER, a small horizontal slider, SVID, VGA, and AUX.
- Status Section:** A collection of controls at the bottom:
 - VOLUME: 0
 - CHANNEL: 0
 - MUTED: ☐
 - POWERED: ☐
 - BRIGHTNESS: 0
 - COLOR: 0
 - SHARPNESS: 0
 - CONTRAST: 0
 - TINT: 0

4) The TV GUI generated by ObjectEditor:

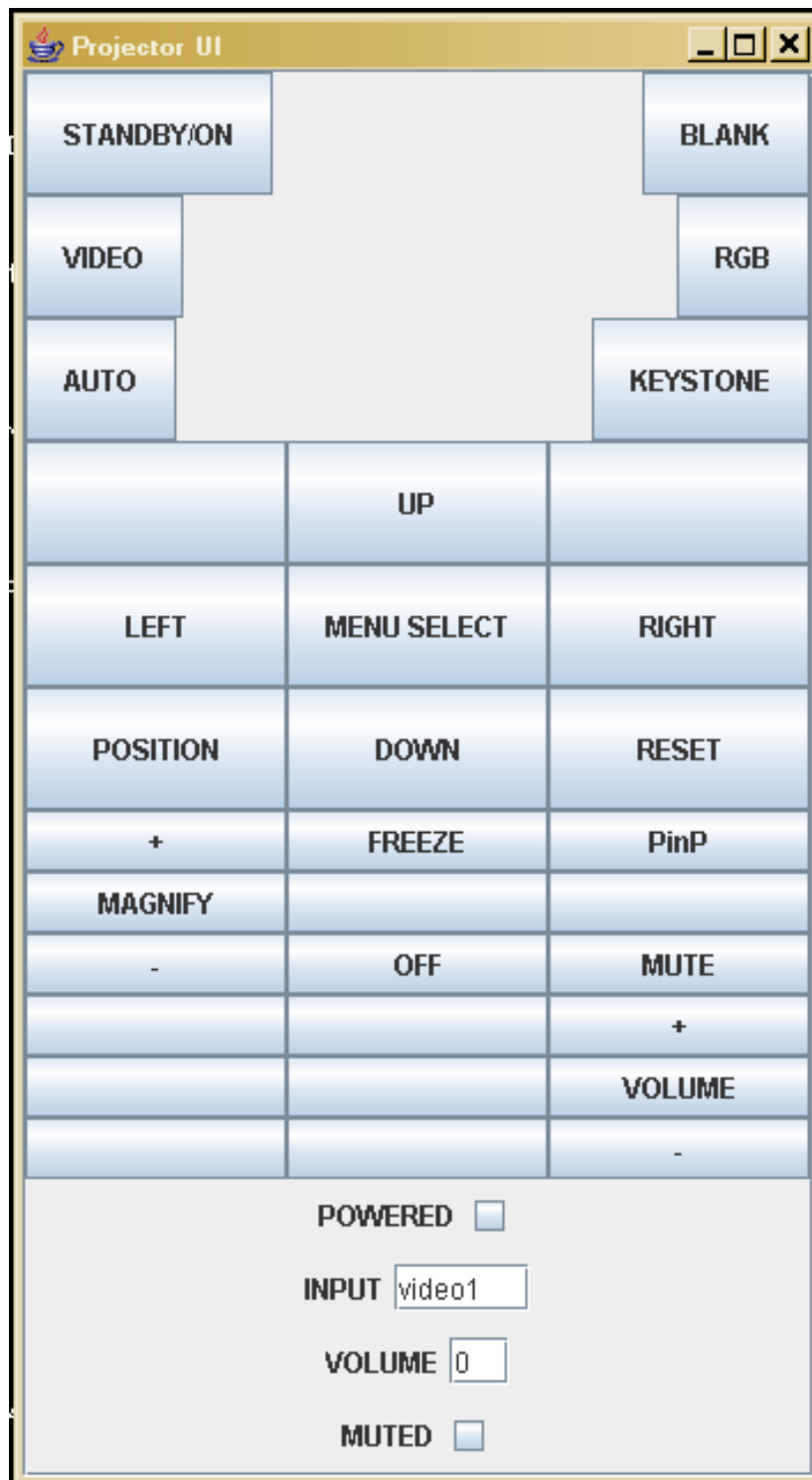
The screenshot shows a window titled "TV@1878144" with a standard Windows-style title bar (minimize, maximize, close buttons). The main content area is divided into two sections. The top section is a grid of buttons for navigating through channels and settings. The bottom section contains controls for adjusting various TV parameters.

A/CH	ADJUST	AUX	CC	CH DOWN
CH UP	DOWN	LEFT	MENU /SELECT	MUTE
POWER	RIGHT	STATUS /EXIT	SVID	UP
VGA	VOL DOWN	VOL UP	ZOOM	0
1	2	3	4	5
6	7	8	9	

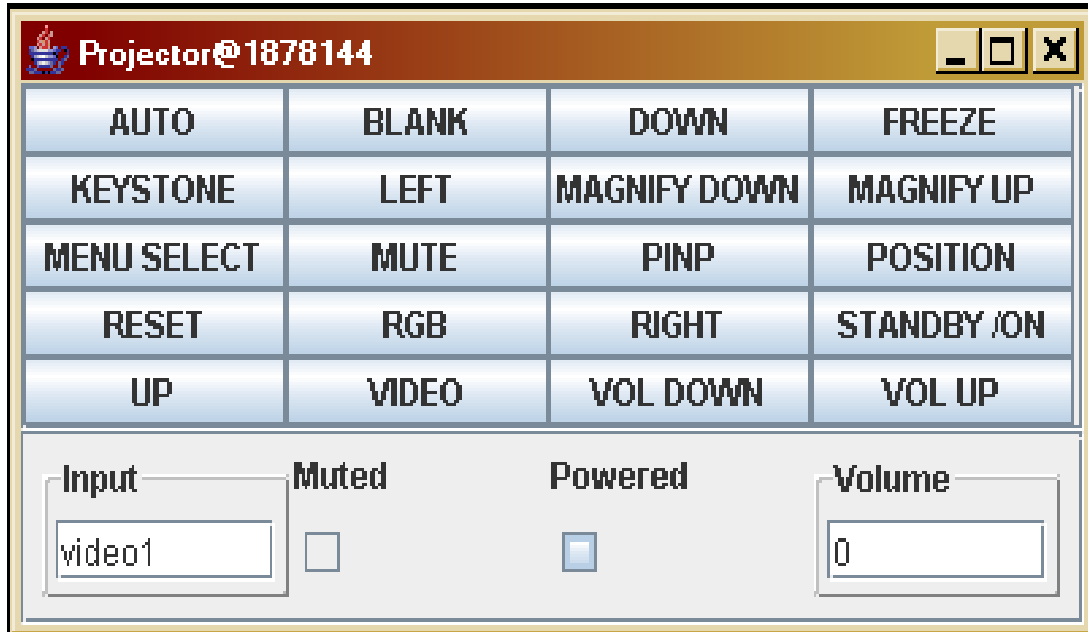
Below the button grid, there are several adjustable settings, each with a text input field showing the value "0":

- Brightness**: Input field with "0"
- Channel**: Input field with "0"
- Color**: Input field with "0"
- Contrast**: Input field with "0"
- Muted**: A checkbox, currently unchecked.
- Powered**: A checkbox, currently unchecked.
- Sharpness**: Input field with "0"
- Tint**: Input field with "0"
- Volume**: Input field with "0"

5) The predefined projector GUI:



6) The projector GUI generated by ObjectEditor:



The screenshot shows a window titled "Projector@1878144" with a standard Windows-style title bar. The main area contains a 5x4 grid of buttons with the following labels:

AUTO	BLANK	DOWN	FREEZE
KEystone	LEFT	MAGNIFY DOWN	MAGNIFY UP
MENU SELECT	MUTE	PINP	POSITION
RESET	RGB	RIGHT	STANDBY /ON
UP	VIDEO	VOL DOWN	VOL UP

Below the grid is a control section with four items:

- Input:** A dropdown menu currently showing "video1".
- Muted:** A checkbox that is currently unchecked.
- Powered:** A checkbox that is currently checked.
- Volume:** A numeric input field currently showing "0".

7) The predefined VCR GUI:

VCR UI			
			POWER
REVIEW	TV/VCR	ENTER/OSD	DISPLAY
1	2	3	
4	5	6	
7	8	9	
CANCEL	0	TIMER	
+	+	+	+
START	STOP	DATE	CH
-	-	-	-
PROG.	PROG. CHECK	SP/EP	SKIP SEARCH
REW	PLAY	FF	
REC	STOP	PAUSE	
MENU	OK		
	UP		
LEFT		RIGHT	
	DOWN		
<div> <div>POWERED <input type="checkbox"/></div> <div>CHANNEL <input type="text" value="0"/></div> <div>TAPE IN <input type="checkbox"/></div> <div>TAPE STATE <input type="text" value="STOPPED"/></div> <div>PLAY SPEED <input type="text" value="SP"/></div> </div>			

8) The VCR GUI generated by ObjectEditor:


The screenshot shows a window titled "VCR3@1878144" with a standard Windows-style title bar. The main area contains a grid of buttons. The first 10 rows of the grid are as follows:




CANCEL	CH DOWN	CH UP	DATE DOWN	DATE UP
DISPLAY	DOWN	ENTER /OSD	FF	LEFT
MENU	OK	PAUSE	PLAY	POWER
PROG	PROG CHECK	REC	REVIEW	REW
RIGHT	SKIP SEARCH	SP /EP	START DOWN	START UP
STOP	STOP DOWN	STOP UP	TIMER	TV /VCR
UP	0	1	2	3
4	5	6	7	8
9				

Below the grid is a section with five controls:

- Channel:** A text box containing the value "0".
- Play Speed:** A text box containing the value "SP".
- Powered:** A checkbox that is currently unchecked.
- Tape In:** A checkbox that is currently unchecked.
- Tape State:** A text box containing the value "STOPPED".

9) The predefined receiver GUI:


Receiver UI

SLEEP		POWER	
1	2	3	
4	5	6	
7	8	9	
SHIFT 10		0	ENTER
VIDEO 1	VIDEO 2	VIDEO 3	DVD/LD
TV/SAT	AUX	MD/TAPE	CD
TUNER	PHONO	5.1CH	
A.F.D.	2CH	MUTE	
		VOLUME	CH/PRESET
MODE	ANALOG DIRECT	+	+
CINEMA STUDIO	BASS BOOST	-	-
EQ/TONE		TEST TONE	
—		—	
+		+	
REAR		CENTER	
-		-	

POWERED ☐

INPUT

VOLUME

MUTED ☐

CHANNEL

TONE ☐

BASS

TREBLE

BASS BOOST ☐

rf VOLUME

lf VOLUME

rr VOLUME

lr VOLUME

c VOLUME

sub VOLUME

rf BALANCE

lf BALANCE


rr BALANCE




lr BALANCE

c BALANCE

sub BALANCE

10) The receiver GUI generated by ObjectEditor:


Receiver@c2b2f6

AFD	ANALOG DIRECT	AUX	BASS BOOST	CD
CENTER DOWN	CENTER UP	CH /PRESET DOWN	CH /PRESET UP	CINEMA STUDIO
DVD /LD	ENTER	EQ /TONE	MD /TAPE	MODE
MUTE	PHONE	POWER	REAR DOWN	REAR UP
SHIFT 10	SLEEP	TEST TONE	TUNER	TV /SAT
VIDEO 1	VIDEO 2	VIDEO 3	VOLUME DOWN	VOLUME UP
0	1	2 CH	2	3
4	51 CH	5	6	7
8	9			

CBalance

1

CVolume

1

Bass

0

Bass Boost

☐

Channel

88.1

Input

video

Lf Balance

1

Lf Volume

1

Lr Balance

1

Lr Volume

1

Mode

A.F.D.

Muted

☐

Powered

☐

Rf Balance

1

Rf Volume

1

Rr Balance

1

Rr Volume

1

Sub Balance

1

Sub Volume

1

Tone

☐

Treble

0

Volume

0

11) The predefined DVD player GUI:

DVD UI			
POWER		OPEN/CLOSE	
STOP	PAUSE	PLAY	
<<	>>	<<	>>
TOP MENU		UP	MENU
LEFT	ENTER		RIGHT
DISPLAY	DOWN		RETURN
PLAY MODE	SUBTITLE	AUDIO	ANGLE
REPEAT MODE	1	2	3
A-B REPEAT	4	5	6
V.S.S.	7	8	9
ACTION	CANCEL	0	>=10
<div> POWERED <input type="checkbox"/> DISC IN <input type="checkbox"/> DISC STATE <input type="text" value="STOPPED"/> RANDOM <input type="checkbox"/> </div> <div> AUDIO MODE <input type="text" value="Digital ENG"/> DISC TYPE <input type="text" value="DVD"/> CINEMA MEMORY <input type="checkbox"/> BASS PLUS <input type="checkbox"/> </div> <div> VSS LEVEL <input type="text" value="OFF"/> DIALOGUE ENHANCE <input type="checkbox"/> CINEMA <input type="checkbox"/> PROGRESSIVE OUT <input type="checkbox"/> </div> <div> SUBTITLE MODE <input type="text" value="ENG"/> ANGLE MODE <input type="text" value="1"/> </div>			

12) The DVD player GUI generated by ObjectEditor:

DVDplayer@1878144

A/B REPEAT	ACTION	ANGLE	AUDIO	CANCEL
DISPLAY	DOWN	ENTER	LEFT	MENU
OPEN /CLOSE	PAUSE	PLAY	PLAY MODE	POWER
REPEAT MODE	RETURN	RIGHT	SEARCH BACK	SEARCH FWD
SKIP BACK	SKIP FWD	STOP	SUBTITLE	TOP MENU
UP	VSS	0	10PLUS	1
2	3	4	5	6
7	8	9		

Angle Mode

1

Audio Mode

Digital ENG 3/2

Bass Plus

☐

Cinema

☐

Cinema Memory

☐

Dialogue Enhance

☐

Disc In

☐

Disc State

STOPPED

Disc Type

DVD

Powered

☐

Progressive Out

☐

Random

☐

Subtitle Mode

ENG

Vss Level

OFF

References

1. Nevo for PDAs. http://www.mynevo.com/nevo_pda.htm
2. OmniRemote. <http://www.pacificneotek.com/omnisw.htm>
3. Remote Possibilities. Usa Today.
<http://www.usatoday.com/snapshot/life/lsnap180.htm>
4. 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*.
5. Bonnet, P., Gehrke, J., Seshadri, P. *Querying the Physical World*. in *IEEE Personal Communications*. 2000.
6. Chung, G. and P. Dewan. *A Mechanism for Supporting Client Migration in a Shared Window System*. in *Proceedings of the Ninth Conference on User Interface Software and Technology*. October 1996.
7. Corporation, M., *Universal Plug and Play Forum*.
<http://www.upnp.org/download/Audio-1-2001Feb.doc>.
8. Czerwinski, S., et al. *An Architecture for a Secure Service Discovery Service*. in *ACM MobiCom 1999*.
9. Edwards, W., et al. *Recombinant Computing and the Speakeasy Approach*. in *Mobicom 2002*. 2002.
10. Eisenstein, J.V., J. and Puerta, A. *Adapting to Mobile Con-texts with User-Interface Modeling*. in *Third IEEE Workshop on Mobile Computing Systems and Applications*. 2000.
11. Gamma, E., et al., *Design Patterns, Elements of Object-Oriented Software*, Reading, MA.: Addison Wesley, 1995.
12. Guttman, E. *Service Location Protocol: Automatic Discovery of IP Network Services*. in *IEEE Internet Computing*.
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. Hewlett-Packard-Corporation. Cooltown. <http://www.cooltown.hp.com>
15. Hodes, T. and R. Katz, *Composable Ad Hoc Location-Based Services For Heterogeneous Mobile Clients*. *Wireless Networks*, 1999. **5**: p. 411-427.

16. Hodes, T.D. and R.H. Katz. *Enabling "Smart Spaces:" Entity Description and User-Interface Generation for a Heterogeneous Component-based System*. in *DARPA/NIST Smart Spaces Workshop*. July 1998.
17. Horvitz, E., et al. *Coordinate: Probabilistic Forecasting of Presence and Availability*. in *Eighteenth Conference on Uncertainty and Artificial Intelligence*. 2002.
18. Isbell, C., Shelton, C., Kearns, M., Singh, S., and Stone, P. *A Social Reinforcement Learning Agent*. in *Agents 2001*. 2001.
19. Kindberg, T. *People, Places, Things: Web Presence for the Real World*. in *Submitted to WWW9*: <http://www.cooltown.hp.com/papers/WebPresence.htm>. 2000.
20. Larsson, B.C.a.O., *Universal Plug and Play Connects Smart Devices*. WinHec 99 White Paper (<http://www.axis.com/products/documentation/UPnP.doc>).
21. MacQueen, J.B. *Some methods for the classification and analysis of multivariate observations*. in *The 5th Berkeley Symposium on Mathematical Statistics and Probability*. 1967.
22. Madden, S.e.a. *The Design of an Acquisitional Query Processor for Sensor Networks*. in *SIGMOD*. 2003.
23. Moyer, S., et al., *Service Portability of Networked Appliances*, in *IEEE Communications*. 2002. p. 116-121.
24. Munson, J. and P. Dewan, *Sync: A Java Framework for Mobile Collaborative Applications*. *IEEE Computer*, June 1997. **30**(6): p. 59-66.
25. Myers, B.A. and M.B. Rosson. *Survey on User Interface Programming*. in *Proceedings SIGCHI'92: Human Factors in Computing Systems*. May 3-7, 1992.
26. Nichols, J. *Using Handhelds as Controls for Everyday Appliances: A Paper Prototype Study*. in *ACM CHI'2001 Student Posters*. 2001. Seattle.
27. Nichols, J., et al. *Generating Remote Control Interfaces for Complex Appliances*. in *ACM Symposium on User Interface Software and Technology*. 02. Paris.
28. Ponnekanti, S.R., et al. *ICrafter: A Service Framework for Ubiquitous Computing Environments*. in *UbiComp 2001*. 2001. Atlanta.
29. Roussev, V., P. Dewan, and V. Jain. *Composable Collaboration Infrastructures based on Programming Patterns*. in *Procedings of ACM Computer Supported Cooperative Work*. 2000.

30. Salton, G., *The SMART Retrieval System: Experiments in Automatic Document Processing*. 1971, Prentice Hall.
31. SavaJe-Technologies. SavaJe OS: Solving the Problem of the Java Virtual Machine on Wireless Devices.
http://www.savage.com/products/SavaJeOS_2.0_Whitepaper.pdf
32. Schiffman S., R.M., Young F., *Introduction to Multidimensional Scaling: Theory, Methods and Applications*. 1981: Academic Press, New York.
33. Schlimmer, J. ChangeDisc:1 Sample Service Template For Universal Plug and Play Version 1.0. <http://www.upnp.org/download/ChangeDisc-1.doc>
34. Shneiderman, B., *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. 4th ed. 1998: Addison-Wesley Longman.
35. Stiles, E. Professor Wants to Put Your Toaster on the Internet.
<http://www.sciencedaily.com/releases/2005/09/050923155217.htm>
36. Sun Microsystems, I., *Jini technology architectural overview*.: from <http://www.jini.org>, and Jini network technology <http://www.sun.com/jini/>.
37. 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.
38. Troll, R., *Automatically Choosing an IP Address in an AdHoc Ipv4 Network*, in *IETF Internet Draft*. 1999.
39. Venners, B. How to attach a user interface to a Jini service.
http://www.javaworld.com/jw-10-1999/jw-10-jiniology_p.html
40. Waldo, J., *A Note on Distributed Computing*. 1994.