MAKEIT: Integrate User Interaction Times in the Design Process of Mobile Applications

Paul Holleis and Albrecht Schmidt

Pervasive Computing and User Interface Engineering
University of Duisburg-Essen, Germany
paul@hcilab.org, albrecht.schmidt@acm.org

Abstract. Besides key presses and text input, modern mobile devices support advanced interactions like taking pictures, gesturing, reading NFC-tags, as well as supporting physiological and environmental sensors. Implementing applications that benefit of this variety of interactions is still difficult. Support for developers and interaction designers remains basic and tools and frameworks are rare. This paper presents a prototyping environment that allows quickly and easily creating fully functional, high-fidelity prototypes deployable on the actual devices. With this work, we target the gap between paper prototyping and integrated development environments. Additionally, new interaction techniques can be significantly faster or slower to use than conventional mobile user interfaces. Hence it is essential to assess the impact of interface design decisions on interaction time. Additionally, the presented tool supports implicit and explicit user performance evaluations during all phases of prototyping. This approach builds on the original as well as extensions of the Keystroke-Level Model (KLM) which allows estimating interaction times in early phases of the development with a simulated prototype. An underlying state graph structure enables automatic checks of the application logic. This tool helps user interface designers and developers to create efficient and consistent novel applications.

1 Introduction

Mobile phones have become a ubiquitous computing platform outnumbering desktop computers. A large portion of current mobile phones offer means for third parties to develop custom software for them. Most notably, there are JAVA ME, Symbian OS, and the Windows Mobile platform. Modern phones provide rich ways for interaction, reaching from colour screens, audio output, and keyboard input to gestures, cameras and audio capture. Additionally, more and more such devices include sensors, e.g. for acceleration (e.g. Samsung SGH-E760, Nokia 5500, iPhone). Interaction with physical objects using barcodes is a common feature in many phones and some devices can read smart labels (e.g. the near field communication, NFC, reader in the Nokia 6131). Furthermore, phones can be extended with external sensors connected via Bluetooth, e.g., for GPS, step counting and ECG.

These basic technical capabilities enable developers and interaction designers to create novel interactive experiences using mobile phones in domains such as data access via physical artefacts, context-aware applications and mobile health applications.

J. Indulska et al. (Eds.): Pervasive 2008, LNCS 5013, pp. 56–74, 2008.

Although APIs exist that allow accessing sensor values, it is often a challenge to create sophisticated user interfaces that exploit all these capabilities. In comparison to conventional interaction techniques, there is little established knowledge about how to build compelling applications using these new means. Hence developments often rely on trial and error which can be costly. In most cases, novel experiences require functional prototypes to be built and evaluated. We believe that prototyping and tool support is essential to make this process efficient. Development environments support the implementation on source code level and to some extent the design of the interaction flow (e.g. the NetBeans Visual Editor[1]). There is, however, a lack of tools that support prototyping interactive mobile applications that make use of advanced interaction techniques using internal and external sensors.

Often, the design process is based on paper prototypes after which the actual implementation is started. It is commonly agreed, however, that at least partially working prototypes are essential to efficiently develop interactive applications and to convey and assess new interaction concepts. Including users in this phase is very important for pervasive systems, as show several examples in a special issue on rapid prototyping in IEEE Pervasive Computing [1].

We address the gap between low-fidelity paper prototyping and actual implementations. The MAKEIT framework (short for *Mobile Applications Kit Embedding Interaction Times*) is used to create functional, high-fidelity prototypes for mobile devices supporting advanced interaction techniques. In particular, we focus on the need to easily create and change applications while at the same time providing assistance in keeping projected end user interaction times low. We contribute:

- an integrated development environment for hi-fidelity prototyping of mobile phone applications creating a code framework for the final implementation
- an underlying model based on state graphs validates parts of the application logic and can detect flaws in the navigational structure and suggest alternatives
- an integrated model to estimate task completion times early in the design without needing to deploy a prototype on the actual target hardware platform.

2 Creating Prototypes of Mobile Phone Applications

This section describes the architecture and interface of the development environment that allows quickly and simply prototyping applications for mobile devices. A common screen based interaction process is reflected in the way the MAKEIT tool chain helps designing applications. A state graph data structure represents the possible flow of actions in a program. By creating such a state graph, the designer lays out the functionalities supported by the application, the possible sequences of user actions and the resulting visual behaviour of the mobile device.

Furthermore, the developer is able to adorn defined transitions between states with additional non-functional parameters, such as KLM parameters. The framework then offers the possibility to retrieve predictions of the interaction time of any possible (i.e. defined) sequence of actions by a potential user. These predictions are based on a modelled, deployed version of the application running on a real phone. The system is

[1] NetBeans IDE, Mobility Pack *http://www.netbeans.org/kb/articles/mobility.html*

designed to support a variety of interaction techniques as listed below. Some common ones are directly integrated, whereas others can be customized and easily added. For some of those interactions, a detailed discussion can be found in the paper of Rukzio et al. about physical mobile interactions [2].

- **Media Capture.** Capturing audio and video and storing or potentially analysing it is used in many applications.
- **Visual Markers.** Using the camera in the phone, interactions based on markers can be supported. This includes simple recognition of barcodes but also advanced augmented reality applications (e.g. Rohs [3]).
- **Proximity.** Based on proximity, actions can be triggered or application behaviour changed. One example is scanning for Bluetooth devices, e.g. Nicolai et al. [4].
- **Gestures.** Accelerometers built into phones offer many opportunities for interaction based on movements and gestures.
- **RFID/NFC.** To capture the identity of a tagged object, RFID and NFC (near field communication) provide easy means. To implement physical mobile interactions the identifier can then be linked to further content.
- **Location.** Using GPS or cell IDs are widely used to get information about the user's location enabling location based interactive applications; see for example the MediaScapes project by Hull et al. [5].
- **External physiologic sensors.** ECG, pulse rate and oxygen saturation are some examples of sensors that can be used to create applications acting on and to body signals (e.g. Nuria et al. in [6]).

The overall concept is similar to that of paper prototyping. Typical steps are to start with a picture of a mobile phone with an empty screen and then to simulate pressing a hotkey, prepare another picture and draw content into the screen. Next to allow the user to touch an NFC tag and prepare another screen. This process is continued until all important states have been prepared. This is exactly the way this tool works, eliminating the difficulty of keeping track what picture belongs to what action.

2.1 Generating the Application Behaviour

One part of the user interface presented to the developer comprises an image of a modern mobile phone featuring the standard set of keys and an empty display (see Figure 1). The next iteration of the tool will feature skins for specific sets of mobile devices with different screen sizes and button layouts. All keys can be pressed using the mouse generating events to the framework running behind the visualisation. Next to the phone are several buttons that can be used to simulate advanced interactions with the phone. Examples include simple gestures, taking a picture or touching an RFID tag. Since not all of those actions are supported by all phone models and new types of interactions are added as we speak, this list of buttons is automatically generated from an XML properties file, which can easily be extended. Using the controls provided by the mobile phone and the action buttons, the developer can implement actions with a simple click. This triggers a dialog in which the interface designer or developer can specify what the contents of the display will be after the specified action has been executed. It can be a simple string or a URL/filename of a web page or

an image which is scaled to fit on the screen. Simple drawings can also be made in place, which is especially useful for people working with graphic tablets.

By repeatedly linking actions to visual elements, a linear sequence of screens can be created which represents the execution of a task in an application; however the majority of applications are more complex requiring richer application logic. This motivates the introduction of the state graph in the following section.

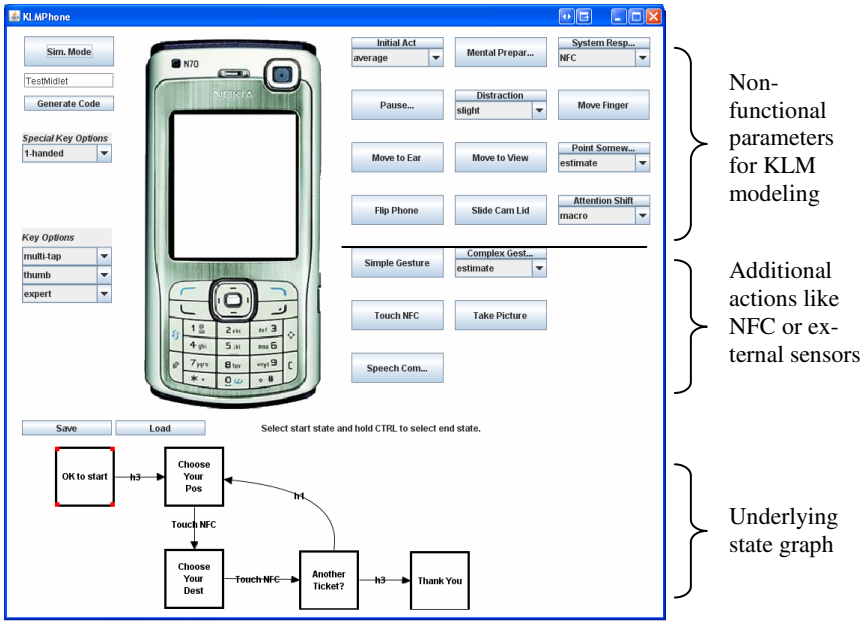

Fig. 1. The keys in the simulated phone and additional interaction techniques can be chosen and the content of its display is controlled by the system

2.1.1 The State Graph

To be able to have a representation of the application logic, we define a state graph $G = (S, A)$. The states of the application that is currently designed represent the set of nodes S. There is an edge $a \in A$ between two nodes S_1 and S_2 if and only if an action has been defined that lets the application switch from S_1 to S_2. All edges are directed from their source to their target node. An edge is also called a *transition* since it describes the transition from its source to its target state. One node can be the source or target of several actions. However, the graph must fulfil the following constraints:

- **Disambiguation Property:** All actions (a_1, a_2, …, a_n) with the same state S as source must be pairwise disjoint. This means that from one state there cannot be transitions fired by the same action to two different states. Otherwise it would not be clear which strategy should be employed to choose the transition that should be used when the according action is executed. This also implies that between two states no two edges have the same action, eliminating redundancy.

- **Start State Property:** There is a distinguished state called the start state S_S that is a source node, i.e. not the target of any transition. This represents the state the application is in right after it has been started.
- **Reachability Property:** For all states S, there must be a path $p(S_S, S)$. A path $p(S_a, S_b)$ is defined as a sequence of edges that connects S_a with S_b, i.e. a path $p(S_a, S_b)$ exists, if and only if there is an n and edges $(a_0, a_1, \ldots, a_n)$ with $a_0 = (S_a, S_0)$, $a_1 = (S_0, S_1)$, $\ldots$, $a_n = (S_n, S_b)$ and $n \in \{0, 1, \ldots\}$. Thus, there is no state that cannot be reached from the start state by a sequence of transitions. Note that this is not the same as saying that every node must have an incoming edge (just imagine 2 connected components that are not connected to each other).

An interesting aspect in the system is that these properties are ensured by construction and thus cannot be violated. Thimbleby and Gow [7] describe several aspects that can be derived from an underlying graph model. The diameter, e.g., represents the task that needs the highest number of actions. The Reachability Property implies that there is no unused state. Weaker properties like the reachability of one state from one or more others (e.g. a standby mode) can be checked as described later. The graph can also be used to check whether all actions can be undone and how costly this is.

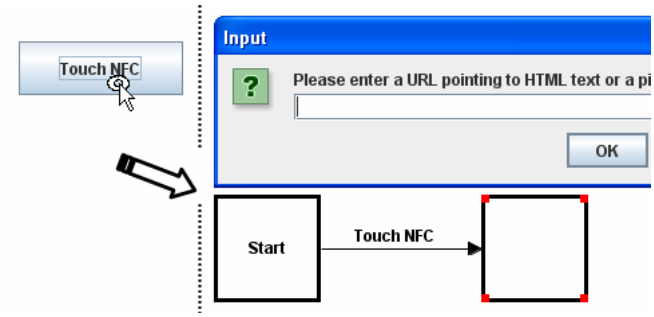

Fig. 2. When triggering the 'Touch NFC' action, a new state is generated and a transition from the start state is added labeled with the action's name

2.1.2 Building the State Graph

MAKEIT provides a visualisation of the set of possible states as well as the transitions triggered by actions. A further part of the user interface presents the state graph described above. Initially, this is only the start state showing an empty phone screen.

The moment an action is triggered, a new node is created in the state graph and an edge is added between the current node and the new node. The edge is labeled with the name of the action (Figure 2). A dialog prompts the developer for the content of the new screen. The new node is automatically selected, indicated by coloured dots in the corners of the rectangle representing the screen of the mobile phone. After specifying the content of the new screen, the next action will continue the sequence and generate another node. This can be used to quickly create a vertical prototype that allows executing defined functionality in detail whereas not all functions that the application will provide when finished are supported.

The creation of the state transitions is not restricted to a linear sequence. When a node of the state graph is selected with the mouse, the defined contents will be updated on the virtual phone's screen and the application is brought into this state. Demonstrating an action can then be done in whatever state the application has been set to. This adds the possibility of leaving a state through different actions. One possible application is to implement different ways to reach the same goal, e.g., press a key, make a gesture or touch a tag. Figure 3 shows the application that the key '8' is used to browse through a list and another one, '5', to activate the selected item.

Adding edges to nodes, i.e. transitions to states, is only limited by the number of different actions allowed for the present state. Following the Disambiguation Property (two edges with the same source node must have different associated actions), transactions that already exist for a specific state cannot create a new edge. Instead, if such an action occurs, the existing transition is fired and the system changes the current state to the target of the edge. Such inputs from the user do not change the state graph. In this way, any sequence of tasks that has already been designed can be walked through and tested. This highly adds to the utility since people often go back to the beginning to recap the task at hand.

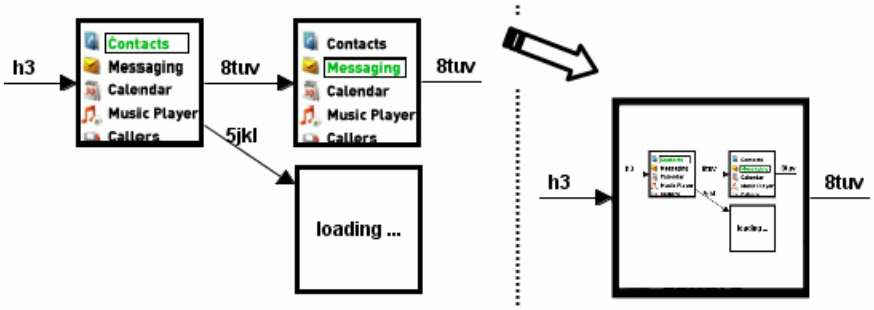

Fig. 3. Reducing the number of visible states by condensing several nodes

2.1.3 Merging States

One of the potential problems with state graphs is that the number of states can grow rapidly. The maximum number of states succeeding a node is only bounded by the number of different actions allowed for this node. However, in our analysis, we found that most applications, besides dynamic screens that are much better implemented in code anyway, do not need many screens. In addition, there are several possibilities to reduce the number of states. One is to **condense** several nodes into a super-node, as is often done to visualise and work with large hierarchical graphs (see Figure 3).

A visually as well as semantically clear approach is based on the observation that applications often return to the same state after different sequences of interactions. Situations in which this occurs afford the merging of equal states. In the case of the visualisation chosen for this project, this means that it must be possible to combine two nodes (shown in Figure 4). We define a **merging** operation $merge(S_1, S_2)$ of two nodes S_1 and S_2 in the same graph as follows:

- for all nodes X such that an edge $e(S_1, X)$ exists, add an edge $e'(S_2, X)$ and copy the properties of e to e'; edges $e(X, S_1)$ is treated analogously
- delete S_1 (and all edges adjacent to S_1, i.e. edges that have S_1 as source or target node) from the graph

The merge operation is defined and executed only if step 1 does not add an edge which would conflict with the Disambiguation or the Root Node Property. By definition, the Reachability Property is not affected by any merge operation.

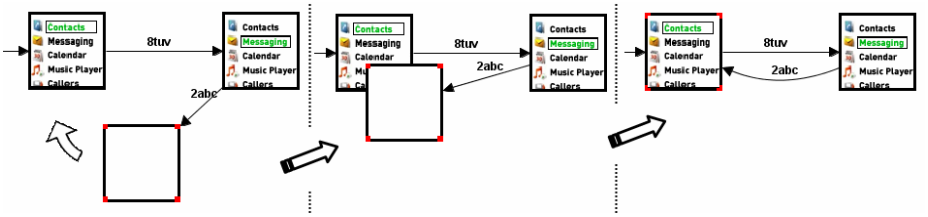

Fig. 4. Merging two states by simply moving one node (empty) over another

Merging states can introduce cycles to the graph which theoretically drastically complicates the automatic calculation of a visually pleasing and planar layout of the state graph. However, in practice, the graphs seem to be fairly easy to layout since most cycles are very short. By moving the nodes in the view, the graph can also be manually adjusted anytime. More importantly, this feature is absolutely essential for many situations like the aforementioned use of a list of items. Scrolling up and down through a list repeatedly generates the same states.

The example in Figure 5 shows a list that can be scrolled by pressing number keys '2' and '8'. The '5' key selects the current item and switches to a state that handles the selected option in the list. This selection method can easily be replaced by, e.g., a gesture without inducing any other change in the graph. This example also illustrates that a node can be the target of several edges as the according state can be reached in

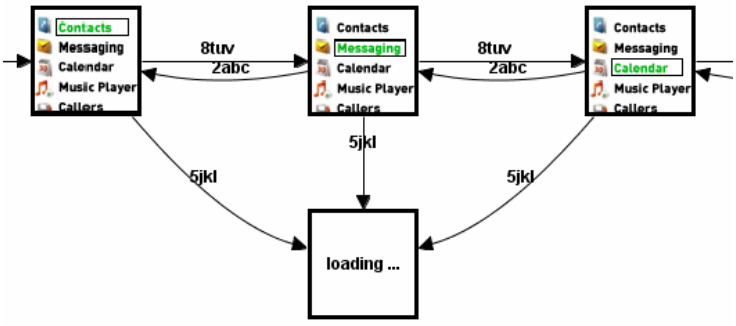

Fig. 5. Designing list scrolling. When an item is selected (key '5'), the same state is reached. Coding will then be employed to show a dynamic screen.

several ways. It also keeps the number of states low by having only one state ('loading …') that is responsible for displaying a reaction to the selected option. One could also split the node way that pressing the execution key will lead to a different state for each menu entry. Any combination of the two approaches is also possible.

Another example for having several transitions to one state is an exit or error state. Applications may have a dedicated exit state reachable from several points in time. Anytime an error occurs, an error state can be reached which offers fallback solutions. The approach can in general *not* be used, however, for a generic message state (presenting, e.g. a message like "This action is not yet supported") since in most cases the application flow should return to the state that initially triggered the message. This would contradict the Disambiguation Property.

We emphasise at this point that neither the state graph itself nor the tools to create it claim or want to be a full-fledged visual programming language. The idea is to leave the handling of difficult tasks to the places where it can be done best: the source code of the mobile application. By omitting any data exchange between states, the available design space is clearly defined.

The design space of the applications that can be created by using this mechanism only is clearly limited. For example, information cannot directly be passed from one state to the next, and it is not known which steps led to a certain state. Although features like that could be added by using a richer data model, the simplicity of the chose approach suffices to quickly start with and concretely test ideas and different interface and interaction designs. In [8], we added touch sensors to the standard keypad of a mobile phone. Using the MAKEIT framework, we were able to quickly develop and test several variations of a contact list showing preview information when the selection button is touched or an image gallery with zooming by touching. The contact list application with a list of four names, e.g., needs only 4x2 states.

Our approach is to separate components through a defined and communication layer. We deliberately decided for this approach and not for plug-in components, as it is more appropriate for distributed pervasive systems.

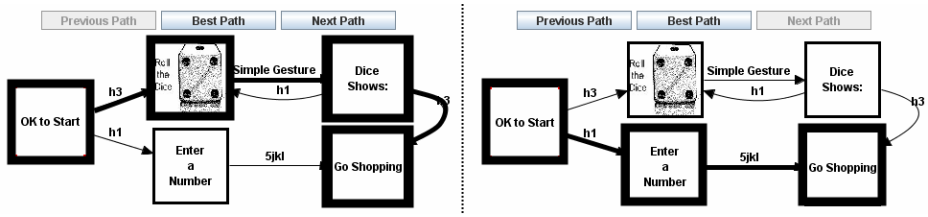

Fig. 6. Two paths from the node 'OK to Start' on the left and the 'Go Shopping' node on the right are highlighted

2.2 Analysing Tasks During Application Creation

One of the important aspects in designing applications is to see and understand if and in what ways a task can be executed with the proposed design. During the design of the flow of the application, i.e. the creation of the state graph, a path finding algorithm can be employed. Selecting a start state S_S and an end state S_e, an algorithm finds all

possible paths $p(S_S, S_e)$. Remember that a path is defined as a sequence of directed edges that connects one node with another. In this case, a path also may not contain a node or edge more than once. This implies that a path cannot contain a cycle and that the number and length of all paths is bounded by the number of nodes and edges in the graph. Note that a path does not necessarily exist between two arbitrary nodes. On the contrary, paths ending in the root node appear rarely and some nodes will even be sinks, i.e. not the source of any edge in the graph, e.g. a dedicated exit state. Those sinks can only be the target node of the last edge in a path, but no paths will start from those. However, the Reachability Property of the graph dictates that there will always be a path $p(S_S, X)$ from the root node to any other node X in the graph.

In the graph visualisation, a path is shown by highlighting its edges as well as the source and target nodes of these edges with thick lines (Figure 6). As said, there are potentially several paths between the selected states which can all be browsed and highlighted. Beside the mere sequences of actions leading to the desired state, the paths can be used to provide an analysis of non-functional properties. This (and why there appears a 'Best Path' button in Figure 6) is explained in the following sections.

2.2.1 Adding Non-functional Properties

Non-functional properties are all characteristics that are not directly concerned with the semantics of an element. In the case of the transitions in the state graph, this means attributes of an action like the time necessary to execute it, the effort needed, the pleasure generated, or the privacy affected by it. In the following, we concentrate on interaction time characteristics and build on knowledge about the Keystroke-Level Model (KLM) introduced in the first sections of this paper.

We have already seen that by triggering actions defined in the state graph, a task can be sequentially walked through and the state of the mobile phone is updated accordingly. To be able to additionally incorporate actions necessary to use the operator model of KLM, this part is elaborated in the user interface. After a version of the application has been defined using the state graph, the user can switch to simulation mode. The user interface is then extended with several additional actions. These KLM actions can also be easily configured and new elements can be added whenever new types of interaction are added in future phones using a property file.

In simulation mode, the root node is automatically selected. All actions can then be executed as defined in the state graph. Whenever an action is triggered that has no according edge defined in the state graph, the action is ignored and a warning is issued. Furthermore, the additional actions in this mode can be used at any time, in any state, in any order. Most of these actions have been introduced in one form or the other in the introductory section on the mobile phone KLM we developed. Table 1 gives a quick overview over the meaning of some of the standard operations, see [9]. The general idea of those operations is that additional information about how a task is executed can be gathered and stored. The mentioned actions mostly concentrate on interaction times. Additional options can be specified to calculate interaction times for standard key presses (one thumb or two finger input, multi-tap or predictive methods like T9, novice, average or expert typist).

As a simple example scenario, consider a poster that displays some products and advertises a URL. The task is simply to browse to this given website. A designer

Table 1. Some non-functional operations supported in simulation mode

Initial Act (average, self initiated, …)	Time necessary to retrieve and look at the phone
Mental Preparation	Time to mentally prepare for the next action
System Response	The time the system needs for computations
Pause	Interrupt for some amount of time
Distraction (slight, strong)	Actions done while being distracted are slowed down on average by some factor
Move to Ear / Move to View	Time needed to move the phone between a state looking at the screen and one close to the ear
Point Somewhere (estimate, Fitts' Law)	Time needed to move the phone to a specific point (e.g. to touch a tag there)

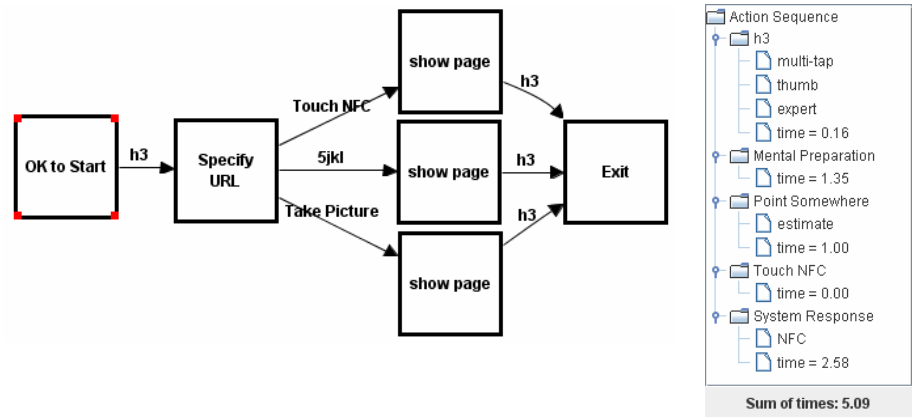

Fig. 7. *Left*: Three different ways of specifying a URL: using an NFC tag entering the URL with the keypad, and detecting a marker using the camera. *Right*: Actions for the NFC interaction from the graph shown on the left.

thinks about implementing one or more of the following three options: enter the URL by hand, take a picture of a marker on the poster or use the phone's NFC capabilities to retrieve the URL from a tag embedded in the poster. A simple state graph that is generated in less than two minutes is shown in Figure 7, left. Since one exit state has been attached to all three interaction methods, selecting the start and the end state will list all three interaction paths.

As next step, the details for each path can be demonstrated. In simulation mode, a separate window shows the action sequence of the currently highlighted path. Here, from the start state, the hotkey 'h3' is pressed and the system prompts for the URL. The act of touching an NFC tag requires four steps: a unit of mental preparation is set to account for the time needed to prepare oneself for the interaction. In the rather coarse modelling of the KLM, this also includes the vague focussing on the target tag (action 'Mental Preparation'). Next, the movement of the phone is done (action 'Point Somewhere'). After the actual reading of the tag ('Touch NFC'), the system needs some time to process that tag ('System Response (NFC)'), see Figure 7, right.

2.2.2 Analysing the Augmented Path

The times for the described actions in the example are: hotkey (0.16 seconds), mental preparation (1.35), pointing (1.00), touching after pointing (0.00), system response time (2.58). This results in a total interaction time of roughly 5 seconds. Those demonstrated non-functional actions (mental preparation, pointing, touching, system response) have now been added to the respective transitions in the graph and need not be re-entered for future calculations of those transactions. The analysis of the other two interaction techniques results in 9.9 seconds (for a short URL of 25 characters such as those produced by TinyURL[2]) and roughly 6 seconds (for a visual marker). Each path between start and end state can be associated with a usability measure like the time the execution of this path would take in real life. The system can then find the 'Best Path' which will be the interaction method that takes the least amount of time. In this example, the algorithm would suggest the NFC interaction. It should be noted at this point that several of the operations like reading NFC tags always result in the same sequence of KLM operators. Those additional non-functional actions can automatically be retrieved and saved. Missing steps, e.g., an anticipated period of mental preparation, this can be easily added to the transition in question.

It is also important to see that the action sequence, augmented with interaction information, can not only be used to compare one sequence to another. We recently used the mobile phone KLM to model different ways of interacting with physical posters. A graphical widget/browser based phone application was tested against one that used NFC tags embedded in the poster. Surprisingly, the model predicted that the text input variant would be considerably faster (2 minutes instead of close to 3 minutes). We ran several tests with different users and found the model to remarkably correct. Interestingly, all users had the false subjective impression that they had been faster with the NFC version. A representation of the modelled sequence of actions is extremely useful to find the parts of the interaction sequence that are responsible for long interaction times. In the scenario under consideration, one of the problems identified was the time lost with checking the feedback of the phone after each single reading of a tag. A proposed solution is that detailed feedback is only given after a series of interactions. This can easily be changed in the state graph of the application by removing the intermediate feedback states and adding a later feedback state.

2.3 Initial User Feedback

To get initial feedback on the prototyping and analysis process, we demonstrated the system to and carried out interviews with 4 experts working in different areas of developing and evaluating pervasive computing applications as well as a couple of students of a user interface master's class. We saw that the main user interface make people try to interact with it at once. Even without initial explanations, they were capable of grasping the idea of generating application logic on the fly. It became obvious that it was not clear that more than linear sequences of actions could be created. After finding out that it is possible to interact with the state graph itself, most people intuitively began to merge states by moving nodes. Actually demonstrating and building the KLM model proved to be more difficult and indicated some necessary refinement in the user interface. All participants saw the

[2] TinyURL service, http://www.tinyurl.com

advantage that the interaction is graphical and that arbitrary screen content could be used. Although the more programming oriented users initially asked for more complex visual control structures, they also agreed that shifting complex things to the source code makes sense. They valued the possibility to quickly create prototypes and provide the starting point for more advanced applications without being hindered to implement whatever they want. The environment helps people concentrate on what they can do best: designers can create and test ideas and interaction sequences and developers focus on the coding. The fact that screens could also be drawn in some separate graphics program was valued especially be the design oriented people since when creating paper prototypes, they often assemble images and text using their own tools.

3 Implementation

3.1 EIToolkit – General Underlying Toolkit Support

Implementing applications using many different programs, hardware and software platforms, communication protocols, and programming languages is, in general, difficult. To counter that, we started the open source project EIToolkit[3], a component-based architecture in which each component is represented by a proxy-like object called a 'stub'. These stubs translate messages between a general communication area to the specific protocol of the devices and back. Any component can then register to listen to messages directly addressed to it or broadcast to all. This enables exchanging components on the fly. The system also allows changing the protocol of the messages on a per component basis. The toolkit currently supports a simple proprietary format over UDP or TCP as well as OSC[4] and RTP[5]. The last two are widely used protocols for audio and multimedia systems and streams. Several microcontroller platforms can be connected through existing stubs as well as over a serial connection. Sample stubs are available, e.g. for the media player Winamp or direct MIDI output.

Independently of the MAKEIT application, we integrated KLM semantics into an EIToolkit module. It is practically platform-independent and can be used remotely. Specific control messages choose the type of KLM like for mobile phones or those for another set of controls. After that, queries are sent to the stub presenting information of an action. For a key press of the "2abc" button on a mobile phone, a sample message might contain the ID 'KEY_NUM2' and parameters '1 thumb', 'expert'. The KLM stub browses its known elements and, if available, sends an answer containing a time value back to the sender of the query, e.g. the MAKEIT system.

3.2 Data Structure of the State Graph

For the implementation of the state graph and its visualisation, we adapted code from the Gravisto graph visualisation toolkit[6]. The data structure provided by the toolkit has been adopted without changes. Beside the basic features of graphs with nodes and

[3] Embedded Interaction Toolkit (EIToolkit) Project Page, *http://www.eitoolkit.de*
[4] Open Sound Control, OSC: *http://www.cnmat.berkeley.edu/OpenSoundControl/*
[5] Real-time Transport Protocol, RTP: *http://www.cs.columbia.edu/~hgs/rtp/*
[6] Gravisto, Graph Visualisation Toolkit, *http://gravisto.fmi.uni-passau.de*

edges, it provides a mechanism to attach arbitrary data to any of the graph elements (nodes and edges) present in a graph. This data is stored in the form of hierarchically structured attributes of various primitive and composed types. This structure is extremely helpful when several pieces of data have to be managed by the graph. As will be seen in a later section, the graph elements do not only have to store the states and contents of the display, but also much information about the transitions between states. Data about the type of action that triggered the transition as well as detailed information about timing and other model parameters are saved with each edge.

The creation and manipulation tools of Gravisto were adopted to ensure the concordance with the graph properties and to enable additional features like merging states. Also, some visual features have been added to correctly display state images. Gravisto also enables saving a generated state graph to file in a standard graph data format called GraphML[7] which is based on XML and supports custom attributes. A saved state graph can then be loaded without data loss at any time and the connection to the mobile phone visualisation in the user interface is immediately updated.

3.3 Code Generation and Extensibility

The whole semantics of the application is stored in the state graph. Nodes contain the data for states and the contents of the screens. Edges represent actions from one state to another and store information about non-functional parameters associated with transitions. One framework component transforms the state graph into a MIDlet, i.e. a Java program for the J2ME[8] virtual machine which can be compiled, moved to, and run on many modern phones. The created application often needs to be complemented with code changes, e.g. for dynamic screen contents. Thus, project files for the NetBeans Mobility Pack are generated and the program can be extended, compiled and downloaded to a phone and tested there. The manifold features of an integrated development environment can thus be exploited, e.g., syntax highlighting, choosing the target platform and debugging. Of course, this also eases making quick alterations and additions to the code itself like implementing the dynamic content of a screen.

Basically, the state graph is implemented as a set of conditional statements. If an event named a occurs in a state S, the state T is loaded if there is a transition (S, T) labelled with the action a. This is (although not optimal) a common and easily understandable way to program such applications. In a mobile phone application, each screen is represented by an object. We use a custom sub-class of the J2ME class Canvas to write code that can load and draw images as well as render text to the screen. It is a low-level implementation of a screen and can also receive key events from the phone's keyboard (standard keys are treated differently from hotkeys).

Beside number/letter key presses and hotkeys, the current implementation of the framework supports advanced interactions using external Bluetooth sensors/devices as well as those supported by the PMIF framework described by Rukzio et al. in [10]. If interactions with NFC tags have been defined, for instance, code is generated that waits for and acts on the reading of a tag. It is possible to add an ID to the edge representing this action. This transition then is fired only if the ID of the read tag is identical. However, for more complex data stored in the tag, the specification of the seman-

[7] GraphML, file format for graphs, *http://graphml.graphdrawing.org*
[8] J2ME, Java Micro Edition Platform, http://java.sun.com/javame/index.jsp

tics is done directly in the code following the principle that the visual design mode should not be overloaded with functionality. The next iteration of the tool will also be able to take into account differences in hardware and software access, e.g. which libraries are used. Currently, the implementation supports S60 phones.

For any other actions – and this includes actions that the user has specified through the properties file – stubs are generated that leave room for the developer to fill it with the concrete code that implements the action. The code generation component uses several template files that contain method stubs and code excerpts. If necessary, these templates can be adapted and extended to work for new interaction techniques like those presented in [8] mentioned before.

The 3 steps to add such touch functionality to the MAKEIT system are:

- add a new action to the transitions.xml file ("Touch")
- add initialization code and specify templates for the code that will handle the new events; specifying, e.g., "$btKeyOn$" will then automatically be replaced by code found in a file "btKeyOn.template"
- add code that shall be executed in the template files, e.g. "btKeyOn.template"

4 Related Work

4.1 Rapid Prototyping Environments

The first of three categories of related works subsumes all kinds of rapid prototyping and authoring frameworks, tools, or methods that can be used to quickly create prototypes of applications to convey or test ideas. The NetBeans visual designer for mobile device applications follows a state based approach as does our project. However, it is restricted in three aspects. First, it is strictly based on the available components like text boxes and lists and does not allow quickly adding free drawings and designs. It also does not directly support advanced interaction methods like RFID tags and cannot integrate non-functional properties like KLM parameters. Focusing on those projects that support in some way or the other mobile or embedded devices as well as more advanced types of interaction (e.g. those requiring sensor input), some shall be mentioned as representatives with no claim for completeness.

The ECT toolkit (Greenhalgh et al. [11]) provides a consistent shared data space across distributed devices. Programming can be done using a visual paradigm. Another recently introduced tool that follows a state-based paradigm is d.tools (Hartmann et al. [12]), implemented as a plug-in of Eclipse. A blueprint of the device to be prototyped can be drawn and widgets representing hardware buttons, sliders, displays, etc. are placed on the drawing. In another editor, a state graph can be created that specifies into which state the device should be transferred on a specific action (like a button press). To our knowledge these projects do not use underlying models that can be exploited for consistency checks or interaction time predictions. In contrast to MAKEIT which generates code that directly and independently runs on a phone, such approaches can suffer from the fact that the prototype depends on the presence of a PC as common gateway and data store. The same holds for several important and useful physical interaction toolkits like the Phidgets [13] that provide readymade hardware UI building blocks for low cost sensing and control implemented as independent components connected to a

computer by USB. The Stanford iStuff toolkit [14] offers another set of elements like buttons, microphones and speakers with a communication layer based on a publish-subscribe mechanism. VoodooIO from Villar and Gellersen [15] combines a virtual stage in Flash with a physical stage that allows arranging physical components and material. Although we describe the capabilities tailored for mobile device prototyping, the MAKEIT infrastructure can be extended with moderate effort to connect to these powerful tools.

The Mediascapes project [16] belongs to a well known set of rapid authoring tools for context-sensitive mobile applications. It focuses on enabling non-programmers to design, implement and deploy applications running on mobile devices. Similar approaches have been made by, e.g. Sohn and Dey with iCap [17], a visual language using if-then rules and relations between people, places and things to define an application logic. The programming-by-example or demonstration paradigm has been followed, e.g., by Topiary and DENIM. They allow specifying triggers of actions, which are comparable to the actions used in the MAKEIT environment. Topiary [18] concentrates on location-based applications where regions are drawn on a map and action triggers are set. The DENIM project [19] shows similarities to the approach presented here, letting the designer create transitions between states. The integration of conditionals, i.e. actions that depend on the properties of a state is planned; this would reduce the number of states visible at the same time as does the condensing of states in MAKEIT. The system, however, requires the user to learn several types of gestures, is designed for web page generation, is not open and easily extensible for external components and does not integrate well with later steps in the application development process. It will be interesting to see how a planned more powerful visual programming language will influence the power and usability of the system.

4.2 User Models

To be able to formalise factors describing human users, models are being developed that characterise users in one or more facets important the use of certain application. There are a lot of approaches that differ in the level of abstraction and formalism, granularity, precision, and target application areas and domains. A detailed discussion of general aspects in user modelling in the area of ubiquitous computing can be found in a special issue on User Modelling in ubiquitous Computing [20].

In this paper, we concentrate on user models from the GOMS family. GOMS is one of the first and most prominent of such models and has been introduced by Card, Moran and Newell in 1980 [21, 22]. It defines goals which can be reached by using a sequence of operators that identify unit actions; if there are several methods that can be followed, selection rules are used to disambiguate them. Several extensions and variants have been introduced to make the GOMS model more powerful. However, even the simplest form has proved to be of much value when having to choose between several design alternatives (see, e.g., work by Hinckley [23] and John [24]).

The inventors of the GOMS model had a specific focus on modelling *physical* actions and concurrent or corresponding mental involvement. In the beginning, one of its main uses has been to model tasks on desktop computers. The Keystroke-Level Model (KLM) has then been introduced by Kieras [25] to aid in the development of more precise interaction models in such environments. Its operators describe basic

actions like key presses, hand movements between keyboard and mouse, and system response times. A number of projects have successfully validated KLM in many different application areas, for example [26, 27, 28]. In the last years, researchers have also effectively adjusted, extended and updated the original KLM with new operators or different values in order to apply it to different and novel interaction techniques. Manes et al. [29], for example, use it for interactions with car navigation systems. One work extending the original KLM for mobile devices is presented by Holleis et al. in [9]. It adds new operators describing advanced mobile device interactions and modifies some of them for the specific use in mobile phones. This includes standard interactions with number keys and hotkeys as well as novel types of interaction like gestures, visual marker recognition, and reading RFID/NFC tags.

Creating prototypes that allow assessing the usage performance is in general regarded as too cost intensive, especially in the domain of mobile and ubiquitous user interfaces. Providing tools that help to keep track of the expected task completion time is valuable in the design process, as often it is hard to estimate such times. Of course, interaction time is only one of the factors that distinguish one design from another. Still, it can be a decisive aspect in making a justified choice. In situations where users have restricted amounts of time (applications for mobile emergency services or programs used while walking to a station), quick solutions can be a significant advantage. Models such as the KLM make predictions about the time experienced users need to execute specific tasks without any need for actual studies. Hence there is no need to create several functional prototypes that have similar timing characteristics. One of the problems identified to hinder the broad use of such models is the cost of learning and constructing correct models. Creating such models can be very time consuming, error-prone, and different people will come up with at least slightly different models. Tools like CogTool [30] (see below) or MAKEIT are therefore needed to make the use of models more practical.

4.3 Prototyping Tools that Support Underlying Models

Prototyping tools that explicitly support underlying models or semantic checks are hardly available. A range of applications exist that allow incorporating actual user traces into the process of developing a UI prototype. SUEDE [31] and WebQuilt [32] are such examples recording user test data for speech and web UIs, respectively. The major difference to our system is that we do not rely on actual user data but use validated interaction models. This drastically reduces time and cost for reaching decisions regarding projected user interaction times.

Gow and Thimbleby describe MAUI [33], an interface design tool based on a matrix algebra model of interaction. Using finite state machines, he can formally state interface properties using linear algebra. There is currently no support for interaction time analyses or code generation for specific target platforms. A start in providing tool support for developers to model applications is CogTool [30]. It uses storyboards to design an application and then employs a cognitive modelling back-end to generate interaction time predictions. In direct comparison with the MAKEIT environment, one can see that the CogTool provides a visual tool to define advanced user models. In contrast to that, MAKEIT focuses on providing support for the actual implementation by generating source code incorporating non-functional parameters.

5 Summary and Future Work

We addressed the gap between low-fidelity paper prototyping and implementations of mobile phone applications. The MAKEIT framework (*Mobile Applications Kit Embedding Interaction Times*) presented in this paper is used to create functional, high-fidelity prototypes for mobile devices supporting advanced types of interaction. In particular, it focuses on the need to easily create prototypes and aid in evaluating and deciding between different interaction designs. Integrated into MAKEIT is support for a KLM based task completion time analysis of the state graph of the application.

While the state graph eliminates the need to remember the order of paper prototype material, the advantage of paper prototyping to quickly react to unforeseen events during studies remains. The required time is slightly more than for paper prototyping but surely lower than for doing any implementations. Using the state graph approach, several types of errors like creating unreachable states can be avoided. In most design processes, considerations like KLM annotations do not play an important part. We argue that integrating such aspects as early in the process as possible, several changes can be avoided later. An open issue is that changes made to the generated code are not reflected in the state graph and have to be repeated each time the code is regenerated. NetBeans, e.g., solves this by only allowing the user to make minor changes in the code at specific places. This, however, reduces the freedom of the developer.

Future work will further evaluate and improve the concept and user interface with a larger user study and concentrate on simplifying the inclusion of standard controls and widgets in the phone's screen like text input fields and scroll lists. Approaches like those seen in the upcoming Adobe Thermo project[9] aim at exactly this direction by automatically converting drawings of, e.g., a text area to a functional text box.

Acknowledgements

The authors would like to thank Prof. Dr. Franz J. Brandenburg and his research group at the University of Passau, Germany, for developing and providing the Gravisto graph visualisation and editing software.

This work was funded by the DFG ('Deutsche Forschungsgemeinschaft') in the context of the research project Embedded Interaction ('Eingebettete Interaktion').

References

1. Davies, N., Landay, J., Hudson, S., Schmidt, A.: Rapid Prototyping in Ubiquitous Computing. IEEE Pervasive Computing 4(4), 15–17 (2005)
2. Rukzio, E., Leichtenstern, K., Callaghan, V., Holleis, P., Schmidt, A.: An Experimental Comparison of Physical Mobile Interaction Techniques: Touching, Pointing and Scanning. In: Dourish, P., Friday, A. (eds.) UbiComp 2006. LNCS, vol. 4206, pp. 87–104. Springer, Heidelberg (2006)
3. Rohs, M.: Marker-Based Interaction Techniques for Camera-Phones. In: MU3I (2005)

[9] Adobe Thermo project, *http://labs.adobe.com/wiki/index.php/Thermo*

4. Nicolai, T., Kenn, H.: Towards Detecting Social Situations with Bluetooth. In: Adjunct Proceedings Ubicomp 2006 (2006)
5. Hull, R., Clayton, B., Melamed, T.: Rapid Authoring of Mediascapes. In: Davies, N., Mynatt, E.D., Siio, I. (eds.) UbiComp 2004. LNCS, vol. 3205, pp. 125–142. Springer, Heidelberg (2004)
6. Nuria, O., Flores-Mangas, F.: MPTrain: A Mobile Music and Physiology Based Personal Trainer. In: MobileHCI 2006 (2006)
7. Thimbleby, H., Gow, J.: Applying Graph Theory to Interaction Design. In: DSVIS 2007 (2007)
8. Holleis, P., Huhtala, J., Häkkilä, J.: Studying Applications for Touch-Enabled Mobile Phone Keypads. In: TEI 2008, pp. 15–18 (2008)
9. Holleis, P., Otto, F., Hussmann, H., Schmidt, A.: Keystroke-level Model for Advanced Mobile Phone Interaction. In: CHI 2007, pp. 1505–1514 (2007)
10. Rukzio, E., Wetzstein, S., Schmidt, A.: A Framework for Mobile Interactions with the Physical World. In: WPMC 2005 (2005)
11. Greenhalgh, Izadi, H.J.S., Mathrick, J., Taylor, I.: ECT: A Toolkit to Support Rapid Construction of Ubicomp Environments. In: UbiSys 2004 (2004)
12. Hartmann, B., Klemmer, S.R., Bernstein, M., Abdulla, L., Burr, B., Robinson-Mosher, A., Gee, J.: Reflective Physical Prototyping Through Integrated Design, Test, and Analysis. In: UIST 2006 (2006)
13. Greenberg, S., Fitchett, C.: Phidgets: Easy Development of Physical Interfaces Through Physical Widgets. In: UIST 2001, pp. 209–218 (2001)
14. Ballagas, R., Ringel, M., Stone, M., Borchers, J.: iStuff: a Physical User Interface Toolkit for Ubiquitous Computing Environments. In: CHI 2003, pp. 537–544 (2003)
15. Villar, N., Gellersen, H.: A Malleable Control Structure for Softwired User Interfaces. In: TEI 2007 (2007)
16. Hull, R., Clayton, B., Melamed, T.: Rapid Authoring of Mediascapes. In: Davies, N., Mynatt, E.D., Siio, I. (eds.) UbiComp 2004. LNCS, vol. 3205, pp. 125–142. Springer, Heidelberg (2004)
17. Sohn, T., Dey, A.: iCAP: Rapid Prototyping of Context-Aware Applications. In: CHI 2004 (2004)
18. Li, Y., Hong, J., Landay, J.: Topiary: A Tool for Prototyping Location-Enhanced Applications. In: UIST 2004 (2004)
19. Newman, M.W., Lin, J., Hong, J.I., Landay, J.A.: DENIM: An Informal Web Site Design Tool Inspired by Observations of Practice. Human-Computer Int. 18(3), 259–324 (2003)
20. Jameson, A., Krüger, A.: Preface to the Special Issue on User Modeling in Ubiquitous Computing. User Modeling and User-Adapted Interaction 15(3-4), 193–195 (2005)
21. Card, S.K., Newell, A., Moran, T.P.: The Psychology of Human-Computer Interaction. Lawrence Erlbaum Associates, Inc., Mahwah (1983)
22. Card, S.K., Moran, T.P., Newell, A.: The Keystroke-Level Model for User Performance Time with Interactive Systems. Communications of the ACM 23(7), 396–410 (1980)
23. Hinckley, K., Guimbretière, F., Baudisch, P., Sarin, R., Agrawala, M., Cutrell, E.: The Springboard: Multiple Modes in one Spring-loaded Control. In: CHI 2006, pp. 181–190 (2006)
24. John, B.E., Vera, A.H.: A GOMS Analysis of a Graphic Machine-paced, Highly Interactive Task. In: CHI 1992, pp. 251–258 (1992)
25. Kieras, D.: Using the Keystroke-Level Model to Estimate Execution Times. The University of Michigan, Unpublished Report (1993), http://www.pitt.edu/~cmlewis/KSM.pdf
26. Bälter, O.: Keystroke Level Analysis of Email Message Organization. In: CHI 2000 (2000)
27. Teo, L., John, B.E.: Comparisons of Keystroke-Level Model Predictions to Observed Data. In: Extended Abstracts CHI 2006, pp. 1421–1426 (2006)

28. Koester, H.H., Levine, S.P.: Validation of a Keystroke-Level Model for a Text Entry System Used by People with Disabilities. In: Assets 1994, pp. 115–122 (1994)
29. Manes, D., Green, P., Hunter, D.: Prediction of Destination Entry and Retrieval Times Using Keystroke-Level Models. UMTRI-96-37. University of Michigan (1996)
30. John, B.E., Salvucci, D.D.: Multi-Purpose Prototypes for Assessing User Interfaces in Pervasive Computing Systems. IEEE Pervasive Computing 4(4), 27–34 (2005)
31. Klemmer, S.R., Sinha, A.K., Chen, J., Landay, J.A., et al.: SUEDE: A Wizard of Oz Prototyping Tool for Speech User Interfaces. In: CHI Letters UIST 2000, vol. 2(2), pp. 1–10 (2000)
32. Hong, J.I., Heer, J., Waterson, S., Landay, J.A.: WebQuilt: A Proxy-based Approach to Remote Web Usability Testing. ACM Trans. Inf. Syst. 19(3), 263–385 (2001)
33. Gow, J., Thimbleby, H.: MAUI: An Interface Design Tool Based On Matrix Algebra. In: CADUI 2004, pp. 81–94 (2004)
34. Rekimoto, J., Schwesig, C.: PreSenseII: Bi-directional Touch and Pressure Sensing Interactions with Tactile Feedback. In: Extended Abstracts CHI 2006, pp. 1253–1258 (2006)