UbiQit: Supporting the Design of Multi-Device Ensembles

Leith Abdulla  
Stanford University HCI Group  
Computer Science Dept  
Stanford, CA 94305 USA  
eleith@stanford.edu

Tina Chen  
Stanford University HCI Group  
Computer Science Dept  
Stanford, CA 94305 USA  
tinachen@stanford.edu

Abstract
In order to take advantage of the communication capabilities of devices, it is important to enable designers to rapidly prototype applications that require multiple devices to work together towards a common goal. In this paper we introduce an architecture that supports designers by fostering rapid design and iteration and allowing these systems to be deployed outside of the lab. We have built UbiQit to accomplish these goals. In this paper we describe the iterations that resulted in our current design, how we implemented our system, an informal evaluation of our interface, and thoughts on future work.

Keywords
Design tools, physical computing, sensors.

ACM Classification Keywords

Introduction
While working on tools for designers, we have seen a growing need to go beyond prototyping single device systems. Designers want to not only take advantage of the computational power of their products, but the
communication capabilities as well. However, each device may communicate over different wireless protocols and require programs written in different languages. We propose UbiQit, a toolkit for prototyping multi-device ensemble that supports the designer in three ways. First, the interactions are specified in a constrained natural language that enables rapid prototyping of applications. Second, the system can be edited in real time, allowing the designer to quickly modify an action or replace a device on the fly. Third, by storing the application logic on individual devices, the system can be taken outside of the lab environment for long-term deployment in user studies.

Imagine a scenario where a designer can quickly author a system where speakers turn on when you enter the room, and turn off when you leave. In order to truly support designers, the tool should allow for rapid prototyping of this system with RFIDs and then facilitate switching the RFIDs to light sensors, and later to sound sensors. Once this system is built, the designer should be able to deploy the system in a user’s home and collect log data from it a few weeks later.

**User Interface**

The UbiQit interface allows the user to specify the interactions between multiple devices by creating rules much like those for an email filter or Applescript. The menu at the top provides controls that allow users to create or delete rules, start or stop testing, hide or show all the devices that can be used, and add a new device. Once a rule is created, it is labeled, minimized, and displayed as a box in the central workspace. These rules can be rearranged via drag and drop or expanded by double-clicking. Once expanded, the rule builder allows users to specify if/when conditions and any number of ensuing actions they would like to occur. Once the system enters test mode, the real time status of all devices is displayed in the right hand side of the interface. Anytime an event or following action occurs, the status display is updated accordingly. See Figure 1 for reference.

**Previous Iterations**

Our attempt to find the right interface involved several design iterations. Initially, we thought a visual programming environment would be best due to the success of tools such as Lego Mindstorms [1]. In this paradigm, users drag icons representing devices onto a workspace, draw arrows between devices they want to interact and are presented with various options. However, through storyboarding and prototyping, we discovered that this interface was often confusing since interactions between devices are much more complex than interactions within a single device. It was difficult to avoid overloading connecting arrows or adding text and prompts to prevent confusion. Also, when working with multiple devices, the user may often want an event to trigger actions on more than two devices. Specifying this through arrows which normally connect two objects proved to be too abstract in our visual interface.

In order to better understand the mental model of how these types of systems are built, we conducted contextual inquiries conducted with five users where we asked them to describe what they would want to be able to build and how they envisioned they would specify the interactions needed for the system. The most common response we received was to list “if/while” statements and follow with a set of
interactions. Our users rarely drew diagrams or utilized iterative concepts. Additionally, in Pane et. al's [7] work studying how non-programmers approach programming problems, they reported similar results of users using event-based rules much more than any other notation. After implementing the natural language interface, we believe we overcame many of the difficulties we had experienced early on with our visual based interface. Our rules allow for explicit specification of how interactions affect different devices and how discrete interactions affect each other.

Implementation
UbiQit consists of multiple parts (as seen in Figure 2), each with Internet access: the UbiQit Server, the User Interface and the UbiQit clients which are the devices being used in the system. The UI consists of an XHTML document that can be used in any browser. Since our UI's main purpose was to allow for the building of rules using dynamic forms, the flow layouts afforded by the Document Object Model (DOM) enabled us to quickly build this application. Currently all the devices are represented by XML files on a webservers. The UI parses these files and then allows the user to pick any of these devices when building rules.
Once the rules are built and the user indicates they want to begin testing, the UI will contact an UbiQit Server through XML Remote Procedure Calls (RPC). All the rules and devices are transferred as XML snippets. Since the UI needs to constantly communicate with the server, a permanent connection is established. The connection is established just like with asynchronous javascript and XML (AJAX) calls, however, we never close the connection on the server side. This is known as a COMET application, and is very similar to how Gmail is able to update the UI once new email has arrived.

The server and clients are all written in python. However, since all communication is done through XML snippets, as long as the XML can be parsed properly and the rules are followed, the device and server can be built in any language. This is important because not all devices can run a python interpreter. Once the server parses the rules, it sends out commands to all the relevant devices.

The clients currently are all written on python and can run on enabled cell phones, PDAs and physical computing platforms such as our modded gumstix. The purpose of the client is to define a set of inputs and outputs that can be performed as well as to listen for incoming rules from the UbiQit Server. Once a set of rules are downloaded by each device, the devices will be instructed to generate events based on new inputs and/or respond to events with output actions.

**Evaluation**

To evaluate our interface, we created a library of virtual devices and built many virtual systems to see what kinds of interactions our interface supports. For example, we built a system where an accelerometer controls the scrolling of a webpage. We also conducted some informal users studies.

**User Study Results**

1. Users were able to quickly understand how our interface worked and could build rules that made complex systems within five minutes after being introduced to the interface. We hope this means that we managed to map well onto their mental model of the system they were trying to build.
2. Users were able to come up with many more systems by exploring what kinds of rules they can
make using the devices provided, since they are all visible each time you edit a rule. One user started with an accelerometer because he thought it was the most exciting device and built interactions that mapped well onto its abilities, such as turn signals on a bike. Another user wanted music to turn on and off depending on if she was in the room, and explored what kinds of sensors would be the best. This is in stark contrast to our initial contextual inquiry where all of our subjects had great difficulty coming up with systems they would like to build. We believe that once it is clear what devices are available and what capabilities they have, one of the main mental blocks to creativity and exploration is overcome.

3. The language the devices use is important. Depending whether a sensor has the option “door open” or “door opened,” users expected it to do very different things. This is especially true when the action is used in conjunction with an “if/while” clause. A “while door open, lights on” may imply that when the door is closed, the lights are off.

**Future Work**

We have tried to adopt a user-centered focus in our design through informal studies with students and storyboarding. However, we need to deploy UbiQit in a formal setting to better understand three things. First, we need to run design task oriented user studies to verify if our interface has a positive or negative impact on the users’ design practice. Second, we need to build physical systems ourselves to explore not only UbiQit’s threshold, but the ceiling of what applications can be built. Lastly, to better support our target audience of designers, we need to better incorporate features into the system that support them not only in the design stage, but also in testing and analyzing the system for future iterations. Better logging features and optional rules to attach code to the devices can be useful in this attempt.

UbiQit’s design space encompasses multi-device systems that work together towards a common application. Thus, it is essential to enable UbiQit to work with a large number of devices. There are two other types of devices we believe it is possible to achieve this. First, we would like to incorporate devices that can communicate with other wireless protocols Bluetooth and Zigbee offer wireless communication at much lower power needs and it makes them well-suited for wireless systems. Second, not all devices—for example sensors—have a CPU on which to run UbiQit Clients, and thus need a central client to parse the rules and handle events on their behalf. This would come in handy if you had an application involving 50 sensor networked motes and you wanted to build an application based on the aggregated information received from them.

**Related Work**

UbiQit draws inspiration from much previous research on ubiquitous computing and toolkits. Our design space, target audience and interface is what separates us from existing work.

D.tools [4] and Exemplar [5] are tools for designers that enable rapid prototyping and testing of physical components (buttons, sliders, knobs) and sensors (accelerometers, bend sensors, light rangers). UbiQit was created because these tools focused on building single devices. Ubiquitous computing allows designers to create applications using multiple devices and we would like to better support this goal.
The Calder Toolkit [6] and Phidgets [3] both provide a physical computing toolkit with an API and architecture similar to that used in GUI programming. Because our target audience consists mainly of non-programmers we wanted to offer a better UI for prototyping with multiple devices. As a result, we instead offer a constrained natural language interface that allows for constructing rules to specify how a system works.

iStuff [2] provides access to an event heap for programmers to build applications using multiple devices. UbiQit uses a similar idea of event heaps to control application events, however we differ in two ways. First, our visual rule based builder allows for non programmers to build ensembles. Second, the final application does not rely on the server to maintain the stack of events. This can be useful if you want to test your application outside of the prototyping environment.

Acknowledgements
We would like to thank Scott Klemmer, Dave Akers and Björn Hartmann for their advice, encouragement and support of our work.

Example citations


