Ubicorder: A Mobile Interface to Sensor Networks

by

Manas Mittal

B.E. Computer Engineering, Delhi University, India (2006)

Submitted to the Program in Media Arts and Sciences
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2008

© Massachusetts Institute of Technology 2008. All rights reserved.

Author____

Manas Mittal
Program in Media Arts and Sciences
August 8, 2008

Certified by____________________________________

Joseph A. Paradiso
Associate Professor of Media Arts and Sciences
Program in Media Arts And Sciences
Thesis Supervisor

Accepted by____________________________________

Deb Roy
Chairman
Academic Program in Media Arts And Sciences

Ubicorder: A Mobile Interface to Sensor Networks

by

Manas Mittal

Submitted to the Program in Media Arts and Sciences
on August 8, 2008, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

This thesis presents the Ubicorder. The Ubicorder is a location and orientation aware sensor network browsing and interactive visualization system together with *in-situ* event definition and identification (termed EDITY). The mobile browser allows real-time viewing of sensor network data, and enables the user to easily establish correlation between physically observed phenomena and their sensor signature. Based on the observed correlations, events are triggered every time a given set of sensor-value conditions are satisfied. Event rules can then be recursively combined to detect complex physical phenomena. Through a first-use user study, we evaluate the system for its usefulness and usability.

Thesis Supervisor: Joseph A. Paradiso
Title: Associate Professor of Media Arts and Sciences, Program in Media Arts And Sciences

Ubicorder: A Mobile Interface to Sensor Networks

by

Manas Mittal

The following people served as readers for this thesis:

Thesis Reader____

Prof. Steven Feiner
Professor of Computer Science
Columbia University, New York City, NY, USA

Ubicorder: A Mobile Interface to Sensor Networks

by

Manas Mittal

The following people served as readers for this thesis:

Thesis Reader____

Prof. Samuel Madden
Associate Professor of EECS
Massachusetts Institute of Technology, Cambridge, MA, USA

Acknowledgments

Working on this thesis, has been a remarkable experience. It has given me a unique opportunity to think deeply about a single topic, for a sustained period of time. For making all these things possible, thanks is due to many people.

To **Joe Paradiso**, who gave me freedom to explore and pursue new ideas. His encouragement and advice have made working on this thesis a memorable experience.

To my readers, **Steve Feiner** and **Sam Madden**, for their in-depth comments, insights and helping me establish a broader context for this work. Steve was extremely meticulous, and provided a unique HCI perspective to the work. Sam provided big picture ideas and help situate the work. Together, they were the best readers I could have had.

To **Scott Klemmer**, **Bjoern Hartmann**, and the **Stanford HCI** crew, for introducing me to research, showing me how, and giving me the chance to.

To **Patrick Winston**, for always keeping me excited about learning, and for injecting humor and perspective into graduate school.

To **Ted Selker**, for giving me an opportunity, for sharing the joy of building, and starting the ball rolling.

To Media Labbers, **Drew Harry** for help with the interface, **Adam Kumpf** for getting me unstuck with Java.

To **Henry Holtzman** and **Ishwinder Kaur**, for help with the MERL sensor network.

To friends at the Responsive Environment Group, **Mathew Laibowitz**, for all

the help, and the punchlines that kept me going. **Behram Mistree** for funny conversation and 7-11 trips, and **Nan-Wei Gong**, who warned me of dire effects should I step out of the office without finishing, and to **Bo Morgan**, for intellectual insights and animated discussion about AI.

To **Lisa Lieberson** and **Linda Peterson** for being efficient enough to be (almost) never noticeable, but always enabling.

To **Peggy** and **Brian**, for being around the lab when nobody else was, and for reminding me about the journey of life.

To friends, **Brandon Taylor, Winnie Cheng, Ambika Goel, Kyle Buza, Takashi Okamoto, Mariya Barch, Susan Yun and Manu Gupta** for making life fun outside of the lab.

To **Minna Ha**, for being the best friend anyone can have.

Finally, to my **Parents**, supportive, warm and an epitome of affection, who would never wonder why their name was last, when it should have been the first.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Ubiquitous computing [61] propounds a vision of computation woven seamlessly into our everyday life. This work approaches ubiquitous computing from the perspective of sensor networks.

Sensor networks are sets of electronic sensors that can compute and communicate to determine sensor data patterns not visible otherwise. Data thus collected from these sensor networks can "extend our perception of the physical world, in both space, time and modality" [39]. Recent advances now make it possible to augment almost all devices with wireless transceivers and processors.

For any sensor network platform to be effective for ubiquitous computing applications, the system must be versatile enough to be usable in scenarios and applications not originally envisioned by the designer. One approach for providing such flexibility is to make the system easily user programmable. This enables end-users to drive the application scenarios, and to utilize their contextual knowledge about the environment.

Deploying a sensor network for ubiquitous computing involves, first, choosing and deploying sensor nodes, second, programming the system to infer meaningful information from sensor data, and third, utilizing this information for user convenience. Observing and interpreting data from these sensor networks is an involved task, and is often a limiting factor in the applicability and deployability of such networks for ubiquitous computing. Applications are frequently "hard-coded," further restricting use to predefined applications. Such a system does not fully utilize the end-user's context and domain knowledge about the environment and the application.

The focus of this work is on the second and third steps, i.e., assisting the user in programming the system to infer meaningful information from sensor data, and to help users utilize the sensor data. It does so in three ways; first, by helping the user to *discover the sensor network*, i.e., locate the sensor nodes, and the sensors they have. Second, as an aid to *view real-time sensor data*, i.e., view, explore and navigate through large quantity of sensor data, and third, by providing tools to help user *interpret the sensor data*, i.e., by enabling users to define, manipulate, evaluate and tweak inference rules. The system then processes these inference rules, identifying for the user when inferred events occur.

We refer to the first two activities, i.e., discovering and viewing sensor data as *browsing*. The Ubicorder is a mobile device that allows users to view near-real-time sensor data from multiple sensors. Discovery and exploration are facilitated by incorporation affordances such as selection of sensor node by physically pointing to it, and by using context information about the user's orientation and location.

The challenge in viewing sensor data is to make the system *scalable*, i.e., view large amounts of data, from a large number of sensor nodes. One approach to address this constraint to to recognize that the user is interested is raw sensor data only to the extent that it enables inferences of occurrence of events of interest. The Ubicorder therefore provides explicit support to design inference rules that translate sensor data to meaningful higher-level information.

We introduce a grammar and interface to aid in the definition, manipulation and tweaking of inference rules. Borrowing from the programming by example/demonstration communities, the Ubicorder's EDITY (Event Definition and Identification System) allows users to define inference rules by *performing* or *observing* the action, correlating it visually with the sensor data thus observed corresponding to this action/phenomena, and thereby designing a simple rule to infer when such actions in the future. The Ubicorder can then detect these events in the future. The mobility of the Ubicorder allows the user to be at the location of event of interest, and quickly match the inferred action with the ground truth.

Finally, we believe that the Ubicorder will make sensor networks more accessible and useful for a larger community of users. The Ubicorder's browsing system is targeted for ordinary end-users. While EDITY system has a higher threshold for use, we hope that the graphical, easy to use interface will promote experimentation by ordinary end-users.

Contributions

The major contributions of this thesis are :

- A mobile sensor data browsing system that enables easy exploration of sensor data, and in turn, augments human perception over "space, time, and modality" [39].

- A system to enable the user to create inference rules corresponding to data patterns to aid in interpretation of sensor network data.

- An interface that allows such rules to be evaluated, experimented with, and tweaked *in-situ*.

.

Chapter 2

Background

This work builds upon, and draws inspiration from several areas of research. This chapter contrast this work from prior art, and motivates the design of the functionality and features of the Ubicorder.

2.1 Tricorders

The Ubicorder is inspired by the "Starfleet Tricorder," a fictional device from the science-fiction TV series, *Star Trek* [26]. The tricorder is a handheld device that scans an area, and interprets and displays the data resulting from the scans [33, 63]. When pointed in a particular direction, the tricorder uses its sensors (built into the handheld) to scan for virtually *any* information about that direction. Typical scenarios ranged from scanning for "novel life forms" and "energy sources" to the more mundane readings for radiation levels and atmospheric pressure. Similar to the Starfleet tricorder, the Ubicorder *scans, interprets*, and *displays* information about the direction in which it is pointed. However, unlike the tricorder, the Ubicorder does not contain all the sensing abilities within the handheld, but instead gleans such data from locally available sensor network.

Many attempts have been made to instantiate such a tricorder. These efforts can be categorized into two main categories: first, researchers have built new sensors that provide *Star Trek* tricorder-like ability to sense from a distance; and second, projects that use commonly available sensors and attach them to a handheld computer.

In the former category, devices such as portable mass spectrometers have been touted to provide tricorder-like functionality. For example, in their paper [14] in *Science*, the authors term a briefcase sized mass spectrometer that can "scan" mass spectra on ordinary samples as a tricorder. Although this paper describes an early research prototype, with time, such a sensor will become feasible and inexpensive. In turn, interpreting data from such devices and appropriately displaying it to users would become vitally important.

The *TR-107 Tricorder Mark 1* [34] by Vital Technologies Corp., is an example of the latter type of device. This device is essentially a collection of sensors attached to a LCD display. Unfortunately, the sensors only report basic readings from their immediate surroundings. There is an Electromagnetic Field Meter, a Colorimeter and a Light meter. Others have released software that "simulates" the interface of real tricorder, designed to execute on a palm pilot(handheld) [30]. The palm pilot version does not incorporate any environmental sensors, and the software exists primarily as entertainment.

In principle, the primary focus of these tricorder replicas is to either collect or display data. Like the original *Star Trek* tricorder, the Ubicorder collects, displays and "interprets" the data, as discussed later.

This work has evolved from the Responsive Environments (Resenv) tricorder. Developed by the MIT Media Lab's Responsive Environments group (jointly with Josh Lifton, Michael Lapinski, and Joe Paradiso) [43, 40]. The Resenv tricorder is a location, orientation, and network-aware handheld device used to interface in real-time to a wireless sensor network embedded in a surrounding domestic and occupational environment. Physically, the Resenv tricorder uses a Nokia N770 Internet tablet for display and user input purposes, a wireless radio, a 3-axis compass with electronic

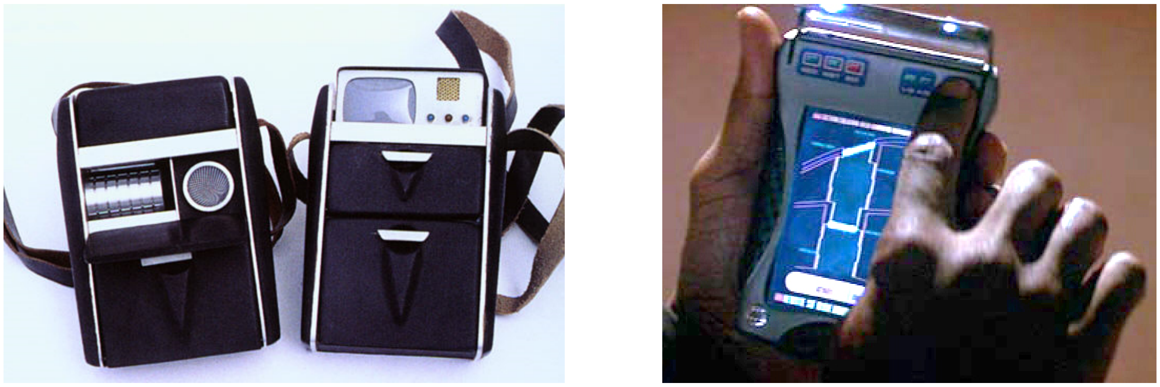


Figure 2-1: The Starfleet Tricorder, (a) The Art of *Star Trek* Version , (b) The Next Generation version. Image ©Paramount Pictures/CBS Studios

gimballing to ascertain absolute orientation in three dimensions (with up to 80^o tilt compensation), a battery pack power supply, and a plastic case to hold it all together. The device knows its approximate orientation thanks to its electronic compass. This, combined with coarse localization (based on Radio Signal Strength Indicator(RSSI)) from nearby embedded sensor nodes, allows for real-time point-and-browse functionality while physically roaming within the sensor network itself.

The Resenv tricorder polls and displays data from the "Plug" [41, 40] sensor network that was deployed on the third floor of the MIT Media Lab. The network comprised of 30 such "Plug" nodes. The "Plug" is a power strip augmented with sensors and a wireless radio. It comes with light, temperature, vibration and sound sensors. Additionally, it also reports the (electric) current and voltage draw from each of its power points.

The Resenv tricorder's data visualization is in the form of icons overlaid over a two-dimensional map of the third floor of the MIT Media Lab. The icons correspond to the "Plug"sensor nodes, and change form to represent sensor data. For example, the icons jitter to indicate vibration, display radial lines that represent light levels, a variable speed rotating needle represents power consumption, concentric circles of varying diameter indicate sound, and the icon color changes if there is a loss of wireless connectivity. An individual sensor node can be selected by tapping on its icon. Doing

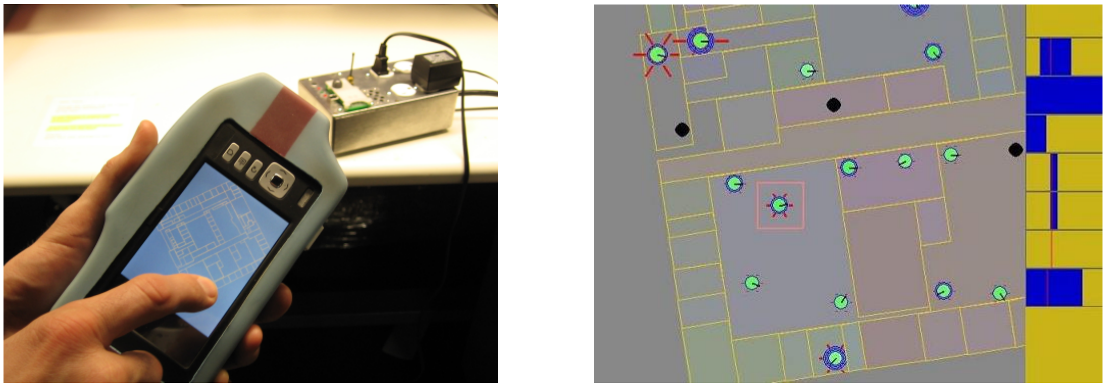


Figure 2-2: The Resenv Tricorder, (a)Form Factor (b) User Interface [43]

so brings up a set of seven bar graphs, each corresponding to a moving average of a single sensor reading (Temperature, Light, Motion, Vibration, Current draw for each of the four outlets).

While the Resenv tricorder scans the area and displays the raw sensor data streams of the selected sensor node, it is difficult to simultaneously monitor sensor data from multiple nodes. Multiple bar charts can be displayed, but the limited screen real estate limits the number of charts that can be coherently displayed. Additionally, Interpreting data from such multiple charts is difficult for users. The changing icons only convey the data coarsely, i.e., the approximate value. More accurate values may be essential for meaningfully inferring form the sensor data.

In order to build a more scalable system, we recognize that the user's interest in the raw data is limited to the extent that such data can be used to infer the occurrence of some event. The Ubicorder therefore provides explicit support to design rules that translate sensor data to meaningful higher-level information.

2.2 Browsing Sensor Network Data

There are varied applications for sensor networks, and these applications drive the constraints with regards to displaying and processing of such data. Some sensor networks are deployed for data collection, with the data analyzed primarily by scientists and engineers. For example, in [47], the authors describe a sensor network deployed to monitor bird habitats. In another example, [32], researchers deployed a network of sensors to monitor the long term health of bridges, and other infrastructure. Such applications often involve no interfaces beyond graphs, and rely on expert knowledge of the user to draw conclusions.

Other sensor networks are designed to be used by ordinary users. For example, "Streetline" [59] is a parking space monitoring system, soon to be deployed in the city of San Francisco, California, USA. The system uses a network of wireless sensors to help car drivers find empty parking spots. While the information has to be displayed in near-real-time and intended for ordinary users, the application is narrowly defined and not readily extensible.

The situation is often different in ubiquitous computing (ubicomp) deployments of sensor networks. Most ubicomp applications require the gleaned information with little or no latency, with the visualization designed to be understandable enough for ordinary users. Additionally, ubicomp targets ordinary users in typical environments (home, office) and a deployed sensor network would ideally be shared across different applications. Therefore, the interface should be versatile enough to be shared between applications. Finally, given the scope of ubiquitous computing applications, the interface must be usable on a mobile/portable device.

The Ubicorder attempts to build such a general purpose sensor network interface designed for ubiquitous computing applications. The realm of general purpose sensor networks, and the interfaces thereof, provide for some interesting comparison with the Ubicorder.

Microsoft Research's SensorMap Project [54] is an example of one such system. SensorMap is a general purpose platform for exposing sensor data culled from a variety of sources. The interface consists of a *Google maps* [20] like map overlaid with icons indicating the location of sensor node deployments. The icons encode sensor value. For example, the color of the car icon changes from green to red to indicate traffic. The

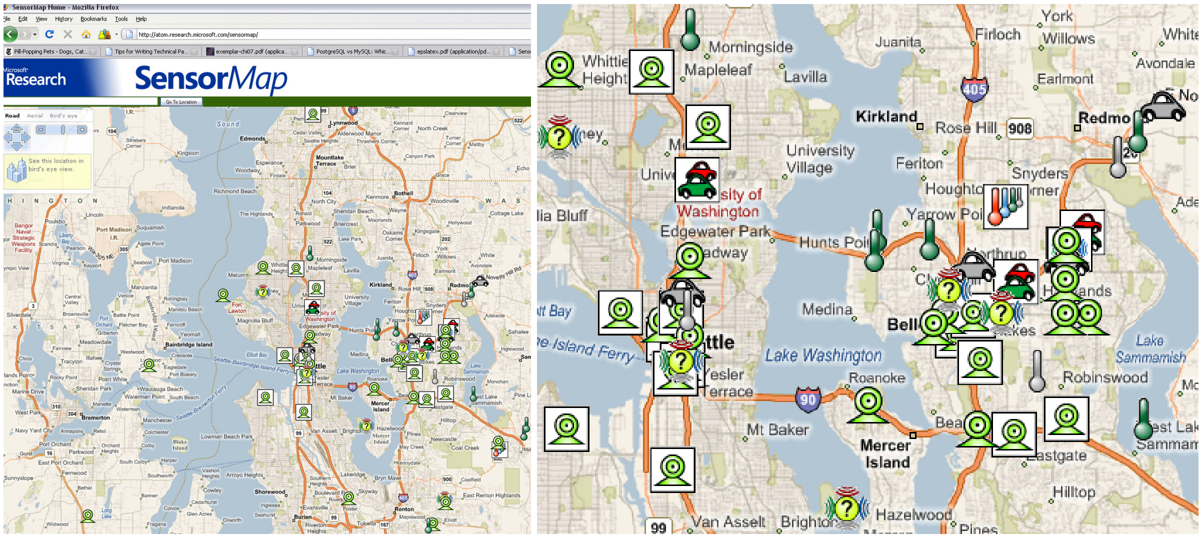


Figure 2-3: SensorMap Interface, (a)Web Page Snapshot(b)Icon Overlay [54]

authors hope that other application developers will build upon this interface and the exposed sensor data, similar to *Google map* mashups (web applications that combine data from multiple sources to create new and interesting applications, for example, a service that pulls "for rent" classifieds from one web site and overlays them on a map) [12] but with real-time sensor data instead. While the interface is for sensors spread out over a few miles rather than indoors, the intent to expose sensor data in order to encourage exploration of sensor data parallels the Ubicorder's objective. SensorMap encourages such exploration by developers. These developers would, in turn, make mashups to be used by the general populace. The Ubicorder's browsing aspect targets ordinary end-users, while the event definition/manipulation subsystem (termed EDITY) targets "advanced" end-users and "sensor network utility workers". . We hope that given the graphical nature of the interface, ordinary end-users will be able to learn to use EDITY over time.

Researchers at Mitsubishi Electric Research Laboratories(MERL) [28] in collaboration with Ishwinder Kaur [31] (later at the Media Lab) have used a network of passive infrared (PIR) sensors mounted on the ceiling to collect data and use it for indoor space usage. A similar setup of such sensors now exists at the Media Lab, and is one of the sensor networks polled by the Ubicorder. The authors also describe a "gestural query interface" where the query is in the form of a path drawn on the map. The system then does SQL queries corresponding to given sensor nodes and displays when the user might have taken the path. Our work differs in several respects. First, the MERL system is tied to a particular type of sensors (PIR movement sensors), in a particular configuration (ceiling-mounted), and for a particular application. Our system, on the other hand, is designed for any set of sensors (binary, discrete and continuous output), placed in any configuration for general purpose examination of sensor data. Second, our interface is designed to allow and encourage quick, end-user exploration of sensor data, and declaration of sensor network templates. Third, the Ubicorder is portable and visualizes near-real-time sensor data, again with an intent to assist the user in correlating sensor data with physical actions or phenomena of user interest. The MERL system, on the other hand, typically runs on large, non-portable displays, and visualizes previously recorded data. Finally, the objective of the MERL work is to support analysis of space utilization using sensor data. Instead, our work aims to support end-users in utilizing sensor network data.

Mobile platforms are often used as sensor network configuration tools. For example, the Great Duck Island sensor network project [47] for habitat monitoring was one of the first systems to use a handheld Personal Digital Assistant (PDA) as a network management tool. Going beyond network management to actual sensor data, in [13], the authors use a PDA to display the availability of nearby conference rooms. Similarly, Maroti et al.'s [48] Sniper localization system uses a handheld as an output device, i.e., to display the Sniper's location as computed by the system.

Finally, the idea of a general purpose sensor network user interface ties well with the idea of Mobiscopes [4]. A *mobiscope* refers to a collection of distributed mobile sensors projected into a taskable sensing system that is able to achieve high-density sampling of a given area through mobility. An individual sensor node may participate in more than one mobiscope. The Ubicorder represents a mechanism to browse the

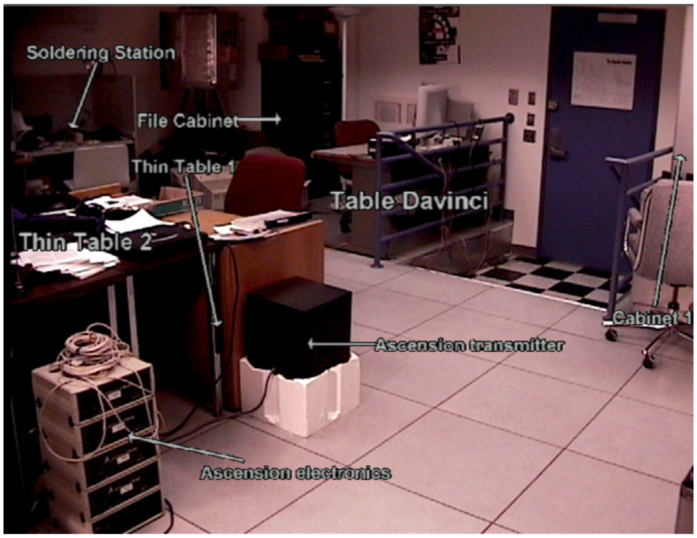


Figure 2-4: Augmented Reality (AR) Browsing [9]

available sensor network infrastructure, and build, deploy, and tweak mobiscopes.

2.3 Augmented Reality

The Ubicorder aims to empower users to interact with the world around them. The field of augmented reality (AR) has similar objectives, and inspired us to work in this direction. AR deals with "augmenting" the real world with computer-generated information. In one approach, a transparent heads up display is used, in which computer generated information and visuals are overlaid with objects. An example of an AR interface is a heads up display system used in military aircraft. For example, information about the horizon is overlaid over the pilot's view.

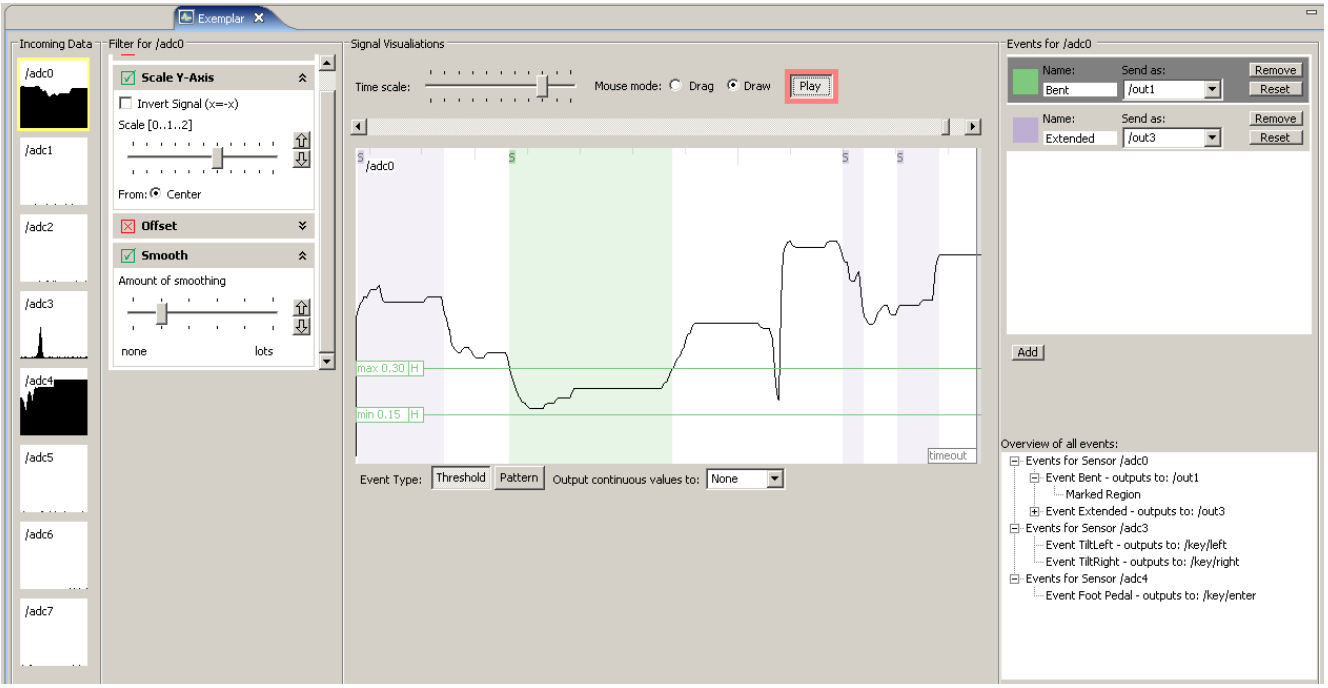
Feiner et al. [18] laid out the foundations of AR as a mechanism to interact with the surrounding real world. In another paper, the authors describe "The Touring Machine," an AR system for exploring urban environments [17]. The system overlays 3D graphics over buildings and locations, presented using a Heads Up Display. The system is location- and orientation-aware (using a differential GPS, magnetometer and inclinometer). The Ubicorder is similar to the Touring Machine in several ways.

It too is location and orientation aware, although it is targeted for indoor applications and ergo uses radio signal-strength-based localization rather than GPS. Additionally, the idea of presenting information co-located with the site it describes is an important motivation for the Ubicorder. AR demonstrates the importance of co-located information and display of that information. In the realm of looking at sensor network data, this co-location of object and information is often ignored. The Ubicorder experiments with this idea.

A more direct inspiration for the Ubicorder came from Jim Youll's work on "Periscope" and "Wherehoo" [64], at the MIT Media Lab. The periscope was a tangible browser for Internet media, built into an old, large format film camera with a LCD display replacing the film back. By panning this camera on a 1D tripod (instrumented with a shaft-encoder), the user could find digital content by physically pointing it towards physical objects and geographical places. The associated "Wherehoo" server binds digital information/media to a real world location and time interval.

There has recently been more interest in the ideas of pointing and browsing of information. Quack et al. recently presented a system that allows users to request information about an object by taking a picture of it [57]. The user can therefore use a camera enabled cell phone to "browse" information linked to physical spaces. The system uses an object recognition method that identifies the object from a query image. The recognition algorithms are assisted by location information (acquired through a GPS). Similarly, Takacs et al. [60] demonstrate an AR system that works by matching an image taken by a GPS-equipped cell phone against a set of location tagged locations.

Finally, we would like to mention that there are several application scenarios for a Ubicorder-like orientation- and location-aware device, outside of the scope of the original *Star Trek* tricorder. For example, researchers at the University of Washington describe an indoor navigation system [44, 25] (for individuals with cognitive



☑ Scale Y-Axis
□ Invert Signal (x=-x)
Scale [0..1..2]
From: ● Center
☒ Offset
☑ Smooth
Amount of smoothing

Mouse mode: ○ Drag ● Draw

Output continuous values to: None

🟩 Name: Bent Send as: /out1
🟪 Name: Extended Send as: /out3

Overview of all events:

```
⊟ Events for Sensor /adc0
    ⊟ Event Bent - outputs to: /out1
        Marked Region
    ⊞ Event Extended - outputs to: /out3
⊟ Events for Sensor /adc3
    Event TiltLeft - outputs to: /key/left
    Event TiltRight - outputs to: /key/right
⊟ Events for Sensor /adc4
    Event Foot Pedal - outputs to: /key/enter
```

Figure 2-5: The Exemplar System [23]

impairments) which uses a tricorder-like location- and orientation-aware PDA. Their system uses computer vision based markers for localization and orientation detection. The Ubicorder could also be used like this device.

2.4 Programming by Demonstration

Programming by demonstration/example systems [15, 38] aim to empower typical users to instruct the system to perform useful tasks. Often the systems involve recording what the user does, and inferring user intent.

The Ubicorder encompasses ideas of programming by demonstration. A typical scenario involves the user performing or observing an action/phenomena, correlating it visually with the sensor data thus observed corresponding to this action/phenomena, and thereby designing a simple rule so that a similar action is detected in the future.

The Exemplar [23] (Figure: 2.4)project comes closest to the programming by demonstration/sensor scripting ideas presented in our system. Exemplar is a programming by demonstration system for scripting sensor interactions. Aimed at designers, the project is intended to lower the threshold for incorporating sensors into design prototypes. A typical interaction is programmed in three steps. First, designers connect sensors to a computer (a set of sensors and interface boards are supported). Next, the user performs the action that they want the system to detect in the future. The interface provides a sensor space visualization of the corresponding sensor signal. Next, the interface enables the user to perform simple conditioning of sensor data streams, such as de-bouncing the signal and introducing hysteresis. The user then defines the rule. The rule could be either in the form of thresholds, or a form of time invariant correlation (Dynamic Time Warping) with a signal template that the user can record. The user can then test the interaction, viewing the results in real-time. The team's CHI paper [23] discusses user study that involves users designing a smart helmet that is augmented with an accelerometer, similar to the Media Lab's smart helmet [58]. The participants author an interaction where tilting the helmet to the side triggers turning on the corresponding blinkers. The Exemplar visual interface is similar to ours in several ways, and is discussed in depth in Chapter 3. This author contributed to the design and implementation of the Exemplar system. The experience has provided valuable design insights for the Ubicorder.

Merrill et al.'s Flexigesture system [49, 50] (Figure: 2.4 is an electronic musical instrument that allows flexible assignment of input gesture to output(sound). Developed in the Responsive Environments Group at the MIT Media Lab, the system is programmed and trained by the user performing the action. The sensor signature thus generated is recorded as a template that is later matched using a form of time agnostic correlation (Dynamic Time Warping.)

Dey et al.'s "a CAPella" [16] is a system to enable end-users to prototype context-aware applications to be by demonstration. This work is related to the Ubicorder

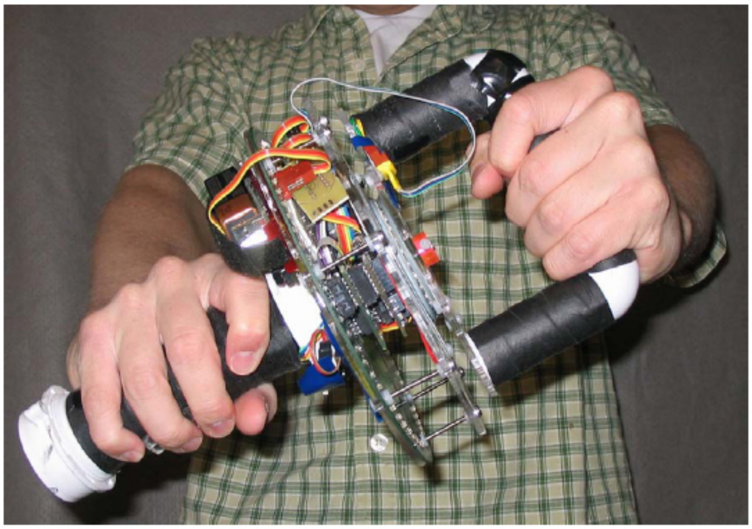


Figure 2-6: Flexigesture [49, 50]

in many ways. First, it makes a strong case for the idea of empowering end-users to create context aware applications, the chief arguments being the user's implicit understanding of the environment. Second, it incorporates the idea of programming by demonstration for context aware applications. The user "marks" the time sections that are relevant, and performs the actions that should be triggered in that scenario. The system creates a corresponding recognizer. Finally, it underscores the importance of an *in-situ* system for creating and editing such rules. The a CAPella system relies on machine learning to extract rules from time series data.

2.5 Query Languages, Stream Processing, and Data Acquisition from Sensor Networks

There have been significant research in the systems community in building new query languages and stream processing engines for sensor network applications. Their work centers around the themes such as reducing latency, increasing computational and power efficiency, and addressing scalability and robustness concerns. The emphasis of their work is not the user interface of the system. The Ubicorder complements this work perfectly: it can serve as the front-end for accessing and using such systems.

Madden et al.'s TinyDB([45, 46]) is a query processing architecture for collecting and organizing data from nodes running the TinyOS [36] operating system. The TinyDB system allows user to write modified form of SQL Queries, and then optimizes the execution of such a query for conserving power.

The Ubicorder also enables the user to graphically define a set of rules to be applied on sensor data. When true, these rules indicate the occurrence of an event. The database community has done significant work in optimizing the detection of events in sensor data streams. For example, Abadi et al.'s Aurora [3] and Borealis [2] present a stream oriented set of operators and optimizations designed for sensor network queries. For example, the *Filter* operator selects the data satisfying a particular condition, and "routes" it based on the conditions it satisfies. Such a system is well suited to be the Ubicorder's rule checking back-end.

Gyllstrom et al.'s SASE [21] presents a system designed for specifying and detecting complex data patterns. While their system is designed specifically for data collected from RFID devices (time, location), their intent is to provide an optimized language correlating and identifying higher order events. The Ubicorder's intent is similar, although it is more generic. A system like SASE could be incorporated as a specific "sensor-module" for the Ubicorder.

The Ubicorder needs to acquire sensor data from the network. Acquisition of sensor data from the network is a complex task, with multiple opportunities for optimization for power and latency. Mueller et al. present SwissQM [53], a virtual machine that presents the sensor network via a single gateway. The system also allows "event rules" to be pushed further down, and optimizes the gateway for a given set of event rules.

2.6 Sensor Scripting

The idea of scripting simple sensor rules based on a combination of sensor values is an old one.

In the gesture recognition community, there have been several projects that incorporate a scripting system. These scripting systems typically allow the user to write a text script. This script defines the sensor conditions, and combinations thereof.

For example, Ari Benbasat, in his masters thesis work done at the Responsive Environments Group (MIT Media Lab), describes a gesture recognition system that incorporates scripting [10] while running on a handheld computer (palm pilot). Individual movements can be defined, and such movements combined through boolean operators. A typical script looks like (from [10]):

```
## Define comparison function
def CompNoDir ( myAlpha, myDuration
, myDirection, theirAlpha
, theirDuration, theirDirection ):
return (abs ( myAlpha - theirAlpha )< dAlpha ) and \
10 (abs( myDuration - theirDuration )< dDuration

## Define Atomic Gestures
## (axis, number of peaks
, alpha, timestamp, width
, direction, matching function)
TwistY = Subgesture (1, 1, 80, 0, 100 , 1, CompNoDir )
LineX = Subgesture (3, 2, 70, 0, 100 , 1, CompNoDir )
dAnyLine = SubgestureDetector ([ TwistY, LineX], 1, gOR )
g0 = Gesture ([ dAnyLine ])
```

```
## Output Functions
def f0 ():
print " Found a straight line "



## Constructing the matching system ( fullgesture , output function )
grs = GestureRecognitionSystem ()
grs. addGesture (g0 , f0)
```

Individual actions are defined by their signal parameters (such as number of peaks, duration of action), and comparators matching such templates with incoming data streams can be defined. However, the textual interface does not focus on ease of experimentation.

In the sensor network community, some work has been done to allow scripting of sensor signals. In [22], the authors describe a sensor network scripting system used for home automation applications. The users write scripts such as:

```
IF movement_detected(sensor-5) == true
  AND lightness(sensor-5) < 800
  THEN switch_power(multi-plug-5, on)
```

Once again, while such systems provide the basic functionality of being able to define detection-actuation rules, they are built on the assumption that the user already understands the sensor data streams. The Ubicorder's portability allows the user to actually go to the location, see the physical event and correlate it with sensor reading being displayed in real time, and then graphically define and experiment with the rules.

The Hive system [51], developed at the MIT Media Lab, tried to provide a unified framework for building building applications by networking local system resources. The system allowed for easy "plumbing" of data and actuation across multiple devices.

The Ubicorder can also serve as an excellent intermediary to such a system, allowing users to configure such connections on-the-fly, and *in-situ*.

Chapter 3

Task Model And Interface Design

This chapter discusses the design of the Ubicorder interface, and ties it to a task model. We also present a walkthrough that demonstrates the Ubicorder's browsing and Event Detection and IDentification sYstem (EDITY). We begin by elucidating the task model and illustrate its relationship to the components of the system. Next, we discuss each component in detail. On occasions, if implementation details influenced our design, we describe them herein.

The task model, Figure: 3-1 presents a high level overview of the facilities and applications of the Ubicorder system, and the interaction patterns followed by the user. The crux of the Ubicorder's functionality can be divided into two parts: the browser, and EDITY (Event Definition and Identification System). The browser enables the discovery of sensor network nodes, and viewing, in near-real-time, the data gleaned from the sensor network. EDITY enables the definition and manipulation of higher-level sensor events. These sensor events correspond to a set of constraints satisfied by a given set of sensor readings, which roughly correspond to a physical/real world action of user interest. These events can then be visualized in the browser, thereby displaying to the user the inferred implication of the sensor data, rather than merely the data itself.

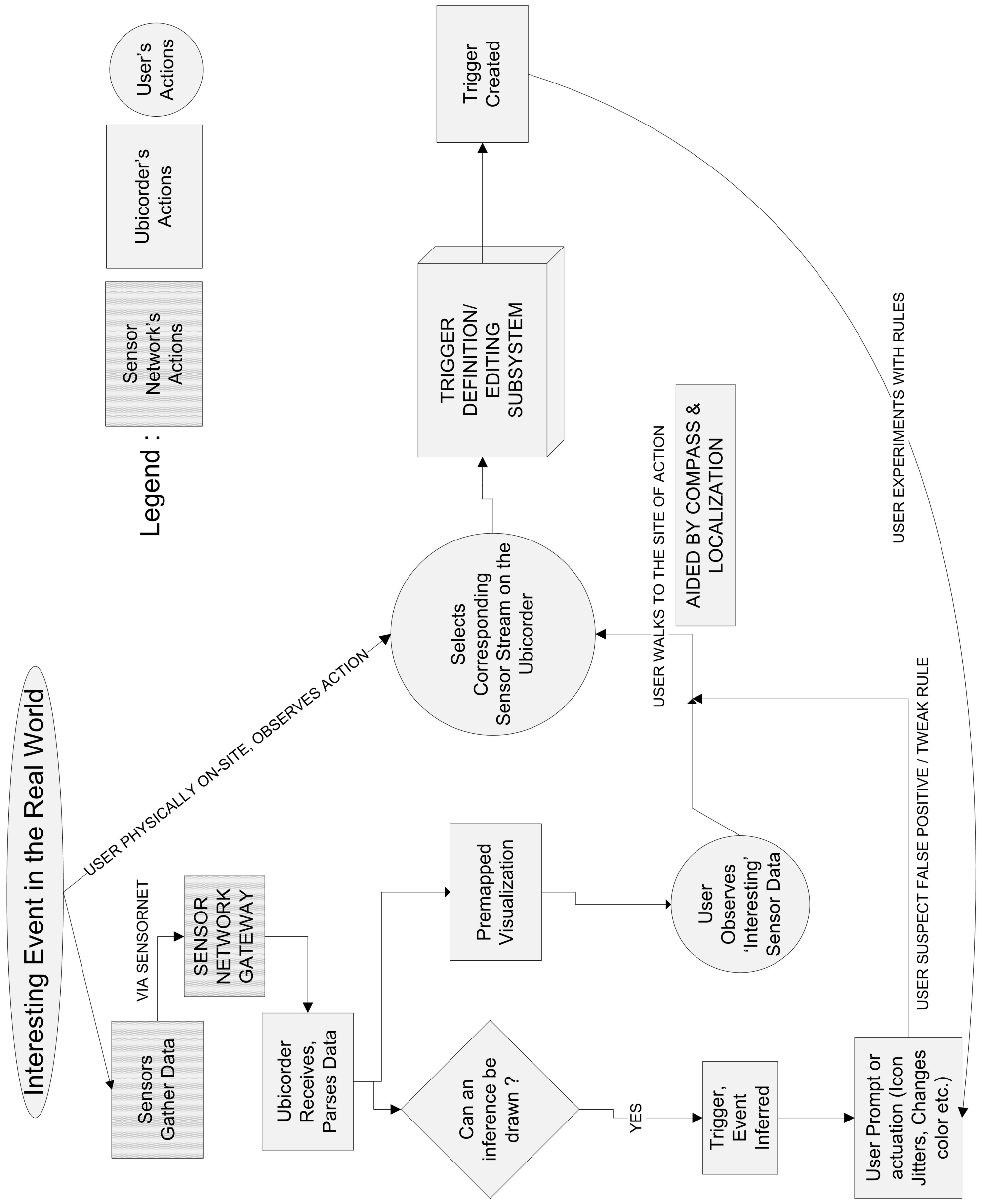

Interesting Event in the Real World
Legend :
Sensor Network's Actions
Ubicorder's Actions
User's Actions
USER PHYSICALLY ON-SITE, OBSERVES ACTION
VIA SENSORNET
Sensors Gather Data
SENSOR NETWORK GATEWAY
Ubicorder Receives, Parses Data
Premapped Visualization
Can an inference be drawn ?
YES
Trigger, Event Inferred
User Prompt or actuation (Icon Jitters, Changes color etc.)
User Observes 'Interesting' Sensor Data
Selects Corresponding Sensor Stream on the Ubicorder
USER WALKS TO THE SITE OF ACTION
AIDED BY COMPASS & LOCALIZATION
USER SUSPECT FALSE POSITIVE / TWEAK RULE
TRIGGER DEFINITION/ EDITING SUBSYSTEM
Trigger Created
USER EXPERIMENTS WITH RULES

Figure 3-1: Task Model : Ubicorder

3.1 Browsing

The Ubicorder's task model begins with the user being interested in some real-world occurrence or happening. Such phenomena may either be observed in person (by being at the location of such an event), or via the Ubicorder's browser. The Ubicorder's browsing mode allows the user to browse near-real-time sensor data through a location- and orientation-aware, map-like interface.

The Ubicorder talks to the two available sensor networks deployed in the Media Lab. The first was a 150 node MERL sensor network that consisted of ceiling-mounted nodes with motion sensors. The second is the "Spinner" sensor network. The "Spinner" nodes sense temperature, light, sound, movement and vibration. The sensor networks, and the facilities they provide are discussed in Section: 5.4.

The user is presented with a floor plan, overlaid with icons depicting sensor nodes. Such an interface is presented on a Tablet PC touchscreen. The tablet is ideally held in the landscape orientation. Figure: 3.1 shows a user holding the Ubicorder. Different icon shapes denote varied *classes* of sensor nodes. For example, in Figure: 3.1, the interface displays the third floor of the MIT Media Lab. Square icons represent movement sensors mounted on the ceiling. Circular icons represent the "Spinner" [35] nodes.

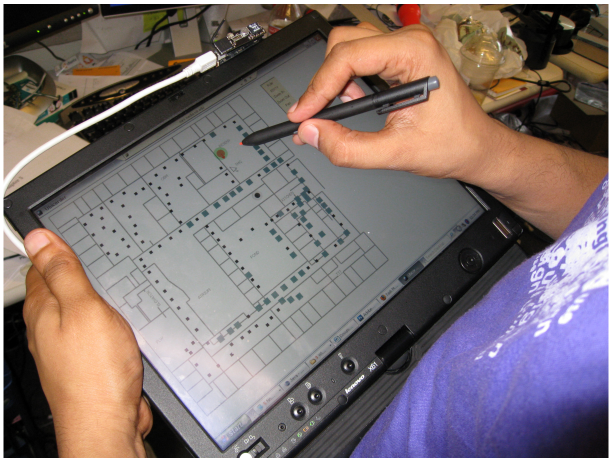


Figure 3-2: Holding the Ubicorder

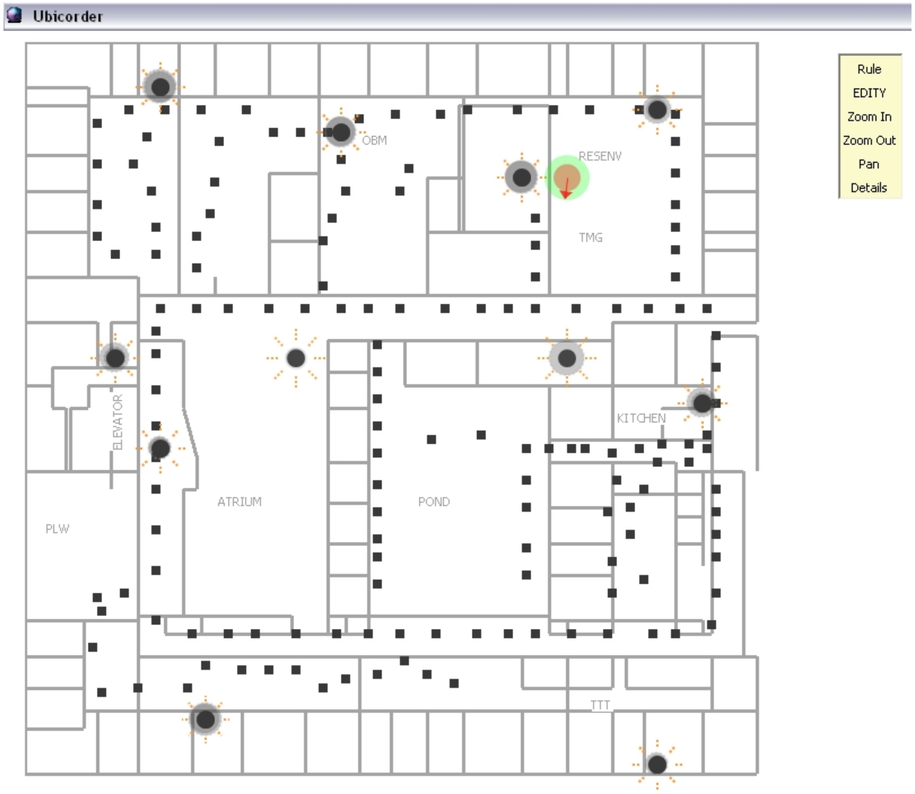


Figure 3-3: Browsing Interface of the Ubicorder. The square icons represent movement sensors, the black circular icons correspond to “Spinner” [35] nodes. The circular icons jitter to convey vibration, change color to indicate temperature, change the size of their halo to represent sound level, change the length of the emanated lines to indicate light levels. The square icons “pop” to indicate motion underneath. The pointed circular icon represents the users location, and increases in size as the localization resolution gets coarser.

Double tapping on an icon (that corresponds to a sensor node) brings up an information panel that displays current sensor data from that node.

Coarse-grained sensor data is conveyed through variations in a node’s icon. The node icons change form or color based on real-time sensor data. For example, increased activity underneath a ceiling-mounted movement sensor is visualized by the node’s icon *popping out* temporarily, i.e., increasing in size and changing color. The circular icons (“Spinner” node) also *‘pop out’* when they register motion. Additionally, a variable diameter external halo surrounding the icon indicates real-time light level readings. Icon changes such as these allow the user a quick overview of sensed the area, and display to the user general trends. For example, a glance would indicate the location of a quiet, well lit place, which might be suitable for studying. Tapping on an icon displays a strip chart of the current sensor data (including recent data). User-defined event rules can also trigger changes in the icon's color or form. We defer this discussion until later (Section 3.2).

The color scheme (gray/black) of the map is so chosen that it "looks dull" when there is no activity or special variations in sensor data. Color indicates the occurrence of something "interesting," e.g., a motion sensor registering movement, or user-defined events (discussed later). Figure: 3.1 shows the movement sensor data pattern corresponding to a person walking through a corridor.

3.1.1 Navigation and Context

The user's approximate physical location and orientation is displayed on the map. This serves as a *cognitive bridge* between the real world and the user interface (UI) by providing to the user *context* of their location with respect to the map on the screen. *Situating* the user simplifies navigation, both on the screen and in the physical space. As discussed in Section: 5.3.2, the location and orientation information is acquired through the Ubicorder's wireless radio(Zigbee), IR receiver and digital compass.

An icon (the "me" icon) indicating the user's location and orientation is overlaid on the floor plan. A directional arrow is placed at the center of this icon indicates the user's orientation. The location and orientation of the icon is responsive to the user's turning and walking around the building. The localization accuracy is variable and depends upon the network's support for Zigbee [7] or IR localization. We convey the locational uncertainty by increasing the diameter of the "me" icon. (see Figure: 3.1.2).

Based on suggestions from a pilot user study, additional labels were placed identifying popular landmarks inside the building. For example, the location of the elevators and the kitchen were marked.

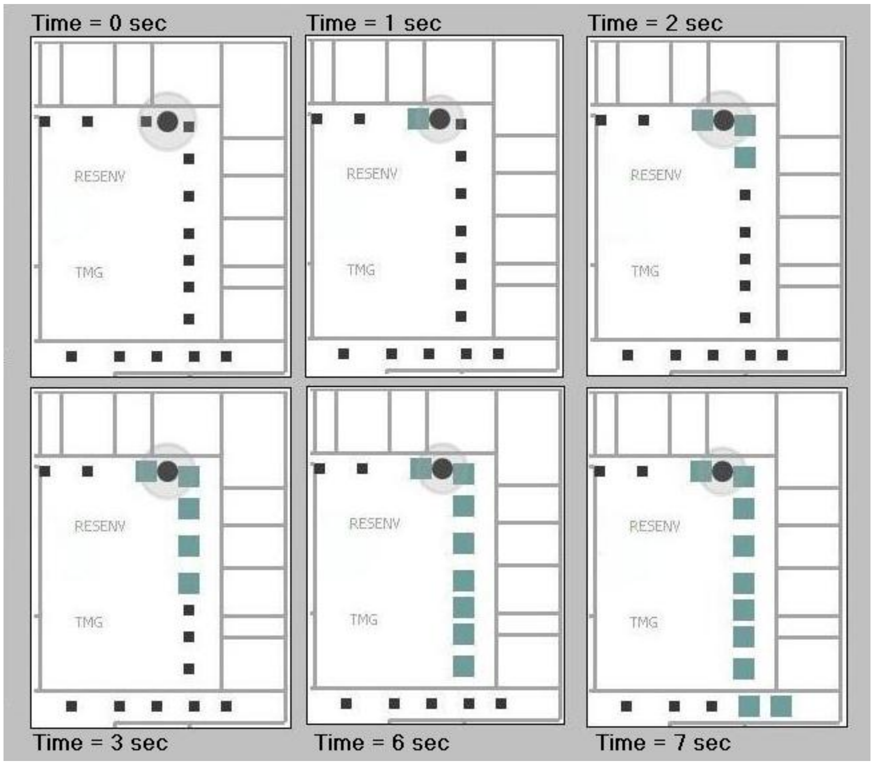


Figure 3-4: "Seeing" a contiguous set of motion sensors detect motion. The six images represent successive time slices with the earliest at the top left and the latest at bottom right. The black icons indicate movement sensors. The icons pop out and change color as they detect motion. Here, it is easy to infer that at least one person is walking underneath (color image).

3.1.2 Point and Select

Physically pointing the Ubicorder toward a sensor node selects it on the User Interface (UI). Selecting the node can also be accomplished by double tapping on its icon. The selected node can then be used for either creating an event rule, or for browsing current sensor data. The intent is to allow users to discover sensor network resources that they are physically close to, and which might prove useful in observing local phenomena.

Once again, the pointing affordance is provided as a *cognitive bridge* between the real world and the UI. As discussed in Section: 5.3.2, the pointing modality is realized by having the Ubicorder read the IR signature emitted by the Spinner node that the

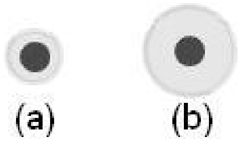


Figure 3-5: Variations in sound level conveyed as variable diameter of the node halo. (a) low-sound level (b) high-sound level

Figure 3-6: Variations in light level conveyed as variable length of lines emanating out of the node (a) low-light level (b) bright-light level

user is pointing towards. The IR is vital here as the RSSI location and orientation estimates are too coarse to be used in proximity. Note that the MERL nodes lack an IR transceiver, and hence can not be electronically pointed at.

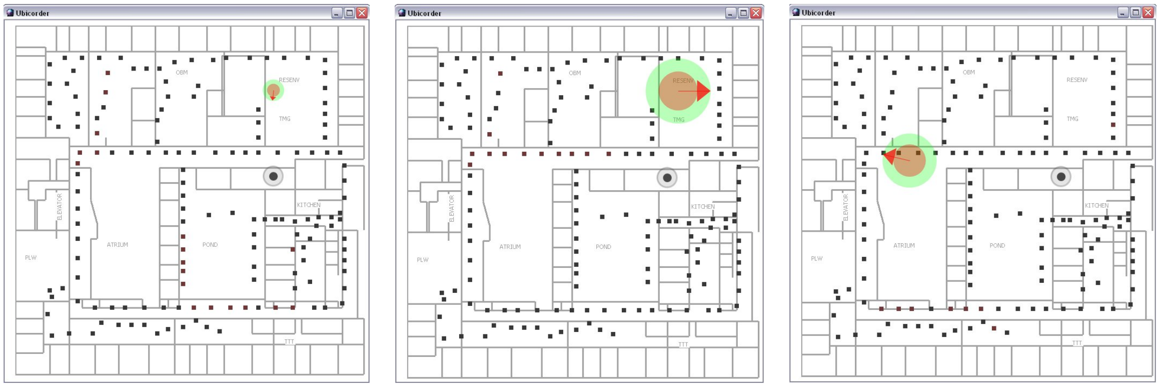


Figure 3-7: The green and red circular icon represents user's location. The icon's arrow indicates orientation, and the size of the icon changes with the localization accuracy.

3.2 EDITY(Event Definition and IdenTification sYstem)

The browsing interface provides, at a glance, a overview of the current state of the area. It is difficult, in this UI, to display quantitatively data from multiple nodes in parallel. However, in order to draw meaningful inferences about the state of the observed area, it is vital to view quantitative sensor data from multiple nodes.

To address these issues, we designed and developed EDITY. EDITY allows users to define, manipulate and test simple inference rules to map sensor data to meaningful higher-level primitives.

3.2.1 Motivation

The Ubicorder is typically used by an end-user for assistance in performing everyday tasks. Since sensor network nodes report raw sensor data, it is left to the user to interpret the received data.

When sensor nodes are used for specialized tasks, considerable effort is spent on data interpretation. Specialized algorithms and data analysis techniques are often used. For example, machine learning algorithms such as Support Vector Machines, Bayesian Inference and Regression Analysis are popular choices.

At the moment, home-office and other ubiquitous computing deployments are unlikely to have the ability or luxury to analyze sensor data deeply enough for making highly reliable inferences. Further, while it may be possible to develop machine learning classifiers for specific scenarios, the diversity of sensor deployments (i.e., their location, calibration and utility) make it difficult to devise a generic- all purpose-inference schema.

The lack of a common classification schema is a well-known constraint in the field of home automation where sensor signals must be manually (and at a high installation cost) mapped to desired actions. For example, in the paper [22], the authors motivate their work, a scripting system for sensor network data, by citing difficulty in programming the system as the main issue.

3.2.2 Overview

EDITY allows users to easily define events, experiment with them, and iterate over their development. The premise is that commonly occurring sensor data patterns that correspond with physical phenomena of interest can be abstracted away as (higher-level) events. Users can design, experiment and iterate over such rules for identification of these events. The system can then, in the future, apply the rules and detect events. Event can be defined in a piecewise and recursive fashion, modeling the human cognitive process and enforcing modularity. Such modularity has several benefits, as discussed later.

3.2.3 Relation with Browsing

Browsing and *EDITY* are intricately linked. It is hard to display large amounts of raw sensor data, or raw sensor data from a large number of sensor nodes meaningfully, in parallel. Also, it is difficult to be able to show historical sensor data in a map-like interface. Further, the user is seldom interested in raw data, but instead in what that data implies. By allowing the user to create implication rules, and detecting them later, such higher-level inferences can be displayed, queried, and stored inexpensively.

3.2.4 Role of Mobility

Being mobile allows the user, at the time of defining the rule, to be physically present at the location of the phenomena, thereby allowing them to correlate the raw sensor data with real events. In other words, the user can literally see the ground truth and the corresponding sensor data.

The interface allows the rules to be crafted in an iterative, hit-and-tried methodology. Using this situated (on-the-spot) hit-and-trial approach has three advantages.

First, it makes understanding the sensor behavior easy. The user can clearly see the correlation between the sensed signal and the real event. There is no prerequisite prior knowledge required about the nature or behavior of the sensor signal. Second, being at the place and tweaking the rule *in-situ* allows for easy testing of such rules. Our interface supports quick modifications of sensor rules, this encouraging the user to experiment and "get it right". Finally, allowing the user to see the correlation between sensor signals, real phenomena, and the rule thus designed, exposes to the user the limitations of the sensing infrastructure, and could prompt the installation of new sensors or relocation of old ones.

3.3 EDITY: Grammar

Rules can roughly be classified as "simple rules," conditional upon a single sensor stream, and "compound rules," conditional upon multiple simple/compound rules. Compound rules are a combination of several simple and/or compound rules combined together via time sensitive boolean operators.

Compound rules can be recursively combined to form other, higher order compound rules. One way to represent this internal rule structure is to think of EDITY rules as forming a directed acyclic graph (DAG), with simple rules forming the leaves of the graph. Figure: 3.3 shows one such DAG.

3.3.1 Definitions

Words in parenthesis indicate usage for the next section (Section: 3.3.2)

- Event: Refers to the occurrence of a physical phenomena/action in the real world that a user is designing the rule for.
- Rule: Defines the set of sensor-value constraints, which, when true, signal the occurrence of an event. EDITY allows the definition and manipulation of these

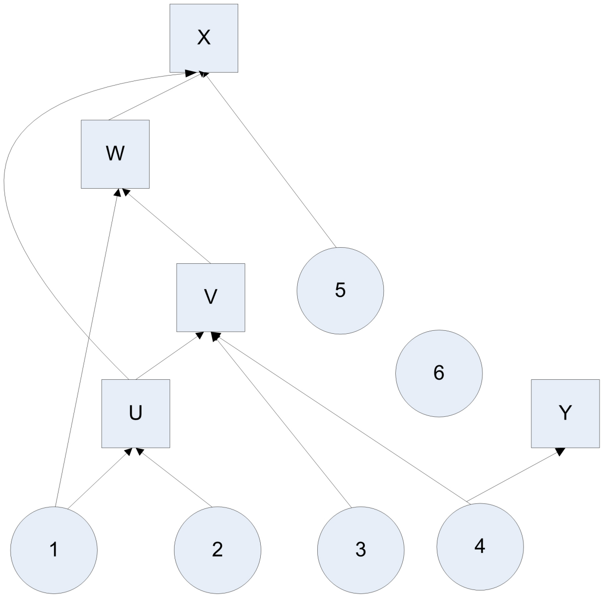


Figure 3-8: A sample Directed Acyclic Graph(DAG) illustrating the structure of simple and compound rules. The circles represent simple rules, while the squares represent compound rules.

event rules.

- Sensor Stream (Ss): Refers to a time series of sensor data values.
- Conditions (Cd): Conditions applied on sensor data stream. Conditions currently supported are maximum and minimum amplitude thresholds.
- Simple Rules (Simple): Check upper and lower bound of filtered sensor amplitude values. Domain is sensor-values, range is boolean (true/false).
- Compound Rules(Compound): Check the state of multiple constituent rules, and combine the states to form one single output. Domain: boolean and *time slack*, range: boolean(true/false).
- Component Rules (Component): We occasionally refer to rules that participate in a compound rule as its component rule. A component rule may be a simple or a previously existing compound rule. Note that component rule is a term invented to describe the system. The Ubicorder's UI does not expose this term.

- Filter (Filter): Filters refer to signal processing operators that act on sensor data streams, and on the output of component rules. The available filters are discussed in more detail in Section: 3.4.2.

- Time Slack (Ts): Introduces time dependency for creating compound rules. If any (filtered) component rule is true within the time slack, the output of the component rule is considered true. Discussed in more detail in Section: 3.5.3.

- Operators (Oper): Operators to combine output of component rules (after filtering and application of time slack). The Ubicorder currently supports the boolean operators AND, OR, and XOR. NOT is supported as a filter, making the operations boolean complete, i.e., any boolean expression may be expressed using AND, OR and NOT operators.

3.3.2 Formal Grammar

This section uses productions to describe the language.

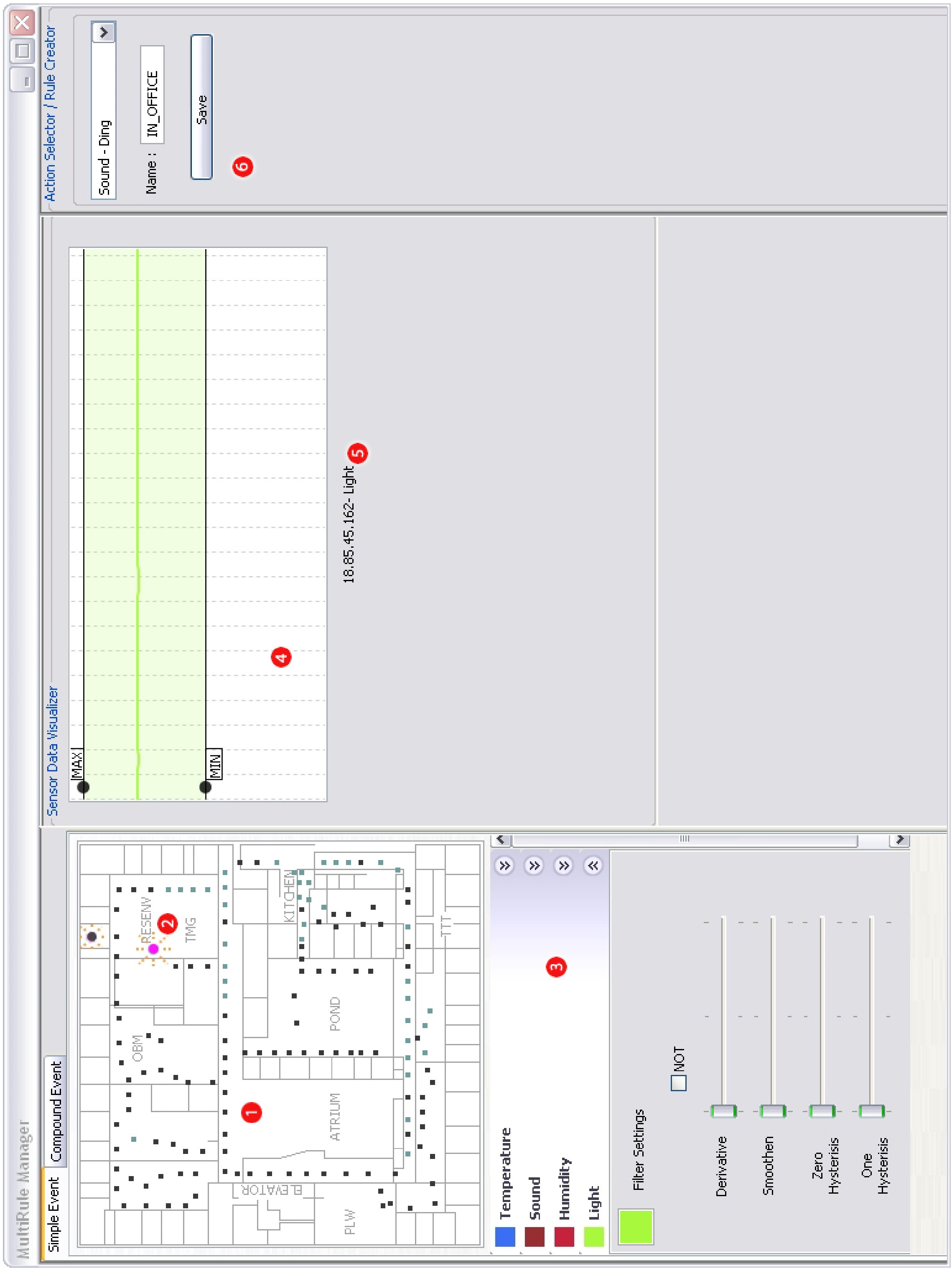
$Ss \overset{Fs}{\rightarrow} Ss$

$Ss \overset{Cd}{\rightarrow} Simple$

$Compound \rightarrow (TsSimple)Oper(TsSimple)$

$Compound \rightarrow (TsCompound)Oper(TsSimple)$

$Compound \rightarrow (TsCompound)Oper(TsCompound)$



Sound - Ding

Name : IN_OFFICE

Figure 3-9: Screen-shot of the Simple Rule Interface (1) Map indicating location/-type of sensor nodes (2) Selected Sensor Node (3) Sensors on Selected Node (4) Data Stream on Selected Sensor, shaded region indicates sensor samples satisfying thresholds, (5) Node, Sensor name, (6) Action: Name of the Rule (IN_OFFICE)and actuation(Sound - Ding)

3.4 EDITY: Simple Rules: Definition, Visualization and Actuation

The process of defining rules can be categorized into three steps:

- Selecting a sensor
- Defining and manipulating decision-rules
- Linking actions and recording simple-rules

In the standard engineering metaphor of data flow, the input port, operation, and output port is placed left-to-right. Our interface follows the same metaphor. The sensor/node selector is in left pane(input), rule definition, manipulation and detection subsystem forms the middle pane (operation) of the UI, and the output, i.e., storing events and or linking actions (output) is place in the right pane.

3.4.1 Selecting a Sensor

The first step is to pick the correct sensor for which to define a rule/event. From the browsing pane, the user clicks the *EDITY* button to go to the "Create Rule" view. The simple-rule interface (Figure: 3.3.2) comes up.

The left pane of the interface allows the user to choose the sensor stream of interest. A browsing window, such as that discussed in Section: 3.1 is the "Node Select" pane which forms the top portion of the left pane. The user selects a node by either physically pointing the Ubicorder toward it, or by tapping on the node's icon. The selected node is highlighted on the map (node's icon is now magenta colored, Figure: 3.3.2(2)). To facilitate the process of screen navigation and selection, the cursor changes to a hand shape when over a selectable node.

Upon selection of a node, a list of sensors exposed by the node is displayed in the lower half of the left pane (Figure: 3.3.2(3)). Clicking on a sensor selects the signal, i.e., expands the sensor name box to display the filters that may be applied to it. Further, the selected signal is plotted on a strip chart in the middle panel (Figure: 3.3.2(4)). The plotted signal is identified by a (user changeable) color indicated in the left panel. The middle panel also displays the textual name of the selected node and sensor (Figure: 3.3.2(5)).

In the simple-rule panel, only one sensor can be selected at any given time. Although, displaying more than one sensor data stream might be useful in certain circumstances, pilot studies showed that it caused ambiguity, especially when rules were being defined. Users were confused with regards to different sensor-values with different units and scales being overlaid on the same strip chart. Selecting a sensor de-selects a previously selected sensor.

3.4.2 Filters

A set of filters may be applied to the sensor data stream. The list of available filters is displayed below the box corresponding to the selected sensor, as in Figure: 3.3.2 and Figure: 3.5.2. Most filters expose a control parameter, k as a slider. Users can drag the slider, and immediately observe the resulting signal (with a changed k).

The list of available filters, and an explanation of the k factor is given in Table: 3.4.2.

Ideally, for a completely expressive interface, these filters should be stackable. However, for the purpose of simplicity, and because the above filters are generally not stacked in arbitrary order, the filters are applied in a predefined stacking order: Not, Derivative/De-Glitch, Smooth, Positive Hysteresis, and Negative Hysteresis.

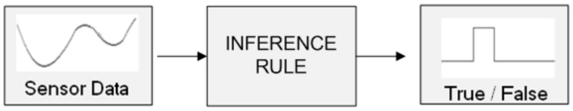


Figure 3-10: Inference Rule, Input and Output

Table 3.1: Filters

Filters			
No.	**Filter Name**	**Description:**	**Parameter *k***
1	Not	Inverts a boolean signal.	Checkbox, True/false
2	Smoothing	Smoothens / low-pass filters the signal by taking a moving average. Useful for removing noise,	k = window size for moving average computation
3	Derivative	Detects rate of change of the signal	Newer $k/2$ samples are subtracted from the older $k/2$ sample, and averaged over $k/2$
4	Positive Hysteresis	Holds the signal at a high value, prevents the signal from dropping fast. Useful as a pulse stretcher. Often used for binary signals	A high value is retained for the next k samples
5	Negative Hysteresis	Holds the signal at the last seen low value. De-bounces the signal. Used for constructs like “Signal should be for at-least 2 seconds. Often used for binary signals	a low value is retained for k samples.
6	De-Glitch	If a signal abruptly goes high or low beyond a threshold and returns to the baseline on the next sample, use the previous sample	k is the maximum difference of a new sample from the baseline that the signal will not be considered a glitch

3.4.3 Defining and Manipulating Decision Rules

The resulting data stream (sensor signal after application of filters) is plotted on a strip chart. The data is displayed in near-real-time, i.e., if the user performs an action to modify the sensor-value, the modified sensor data is visible immediately. The strip chart scrolls through as newer data comes in. The user can also pause the strip chart, or scroll back to recent data. The time length the user can scroll back to see the data is dependent on the size of the data buffer, currently set at equivalent to about ten seconds of data.

The rule creation system works by allowing users to set thresholds on the value of the (filtered) signal. There were a number of constraints on how the rules should be designed.

Below, we describe these constraints and explain why this simple thresholding approach was chosen.

- **Ease of Definition**: It should be relatively easy and quick for the user to define the rule. Our approach relies on "dragging" a pair of horizontal threshold lines on the strip chart. The lower and upper threshold lines correspond to the upper and lower threshold respectively.

- **Quick Evaluation**: Again, in order to permit iterative "sculpting" of sensor rules, the user must be able to quickly see the result of the rule they have created. Further, the user will typically be on-site of the event/phenomena that they are making the rule for, and may or may not be able to spend extended time at the given location. Therefore, machine learning approaches that involve long latency between training and classification are unsuitable.

- **Ease of Manipulation/Modification**: In order to support iterative experimentation of inference rules, it should be fairly easy to modify and experiment with the rule. Most machine learning approaches, for example, *neural networks*, are *opaque* in terms of the classification methodology and are not amenable to "tweaking". Some techniques, such as *decision trees*, do allow for flexibility in the classification methodology but are still non-trivial to manipulate. By using multiple rules, EDITY can generate a classification schema similar to a decision tree.

- **Limited Training Data**: Any user system is constrained in terms of limited quantity of labeled training data. Users can not be expected to label large quantities of training data. For example, consider an inference rule to detect whenever someone walks through a path based on sensor data from movement sensors placed on the ceiling. Labeling the training data would involve explicitly specifying when someone walks through. Relating the sensor data to ground truth, especially when the user also has to perform the action, is cumbersome process. In our simple rule based system, the user labels real data. Further, by exposing these innards of the system, the magnitude of so called training data drops exponentially.

- **Difficulty with time series feature**: The first step in any traditional machine learning approach is feature extraction. Feature extraction is directly related to the nuances of sensor data stream, and therefore, it is difficult to design a generic algorithm for this purpose.

- **Versatility and Extensibility**: The approach must work for different types of sensor data, i.e., binary, discrete and continuous. The thresholding approach works for *all* all data. Further, the filter model allows specifying of more elaborate pattern matching systems. Ultimately, every quantitative pattern matching algorithm has a numerical quantity to define "matching" level. The threshold could be set for these values instead of raw signal values. For example, consider when high "correlation" with a template is required to specify a match. A correlation filter (tied to a specific template signal) can be added in the list of available filters. Upon application of this filter to the sensor stream, the range of acceptable correlation coefficients can be specified as thresholds. Similarly, a Fourier filter could be used, and appropriate cut-off thresholds(frequencies) could be specified. In general, the threshold approach is both versatile and extensible.

Setting the rule using strip chart lines

The middle pane features a strip chart pane where the user-selected data stream is plotted(Figure: 3.3.2(4)). This pane has a pair of user draggable horizontal lines signifying the upper and lower thresholds being set. As the user moves the horizontal lines, the section of the data stream satisfying the constraints are highlighted. Further, the user can "look-back" into the data stream to examine time segments where these constrains would have been satisfied.

3.4.4 Linking Actions, Recording Simple Rules

The final step is to associate the rule to action. The action could be some form of actuation to evoke the user's attention. We currently support actions such as: selecting, or coloring the node's icon (in the map drawn in the browsing view, and in the left panel of the EDITY view), playing an audio file, or sending a key stroke (through the Operating System, using the Java Robots API [29]). The last option allows the user to control external programs. In the interface, a drop down "action" menu lets the user choose among pre-defined actions.

In the present state, the action is either edge triggered or level triggered, depending upon the action. For example, the "ding" sound is forward edge triggered whereas the "Color Node Icon" is level triggered. In order to preserve ease of use for the user, the edge or level triggering is directly associated with the action, i.e. some actions are inherently edge triggered, while others are inherently level triggered. A wide variety of actions which are, by default, either edge or level triggered, are provided.

Alternatively, or in addition to actuation, the user can "save" the rule. Saving implies that the rule (as defined by the node, the sensor of the node, the filters applied and the thresholds) will be evaluated for whenever new data is received from the included sensor. A binary output stream is exposed by such a saved rule. These simple rules form the building blocks for *compound rules*, discussed in the next section. To record a rule, the user gives the rule a descriptive name and clicks on the "save" button. The system ensures that a rule with a same name does not already exist, and if so, saves the rule.

It is worth mentioning that "writing a text name" is the first instance in our **pen based interface** where a keyboard would a be better option than a pen/touchscreen interface. One alternative to having the user enter a descriptive name was to put in a default text name, such as nodename-sensorname-action. The major disadvantage with that approach was the assumption that the system name for the node is same as the explicit representation of the node. For example, most users are likely to identify nodes by names such as “Kevin’s office,” while the internal system representation is likely to be a combination of hexadecimal numbers. Further, giving the user control over the naming process enables the user to inject context and information about the rule in the name. An on-screen touch keyboard is provided for the purpose of entering the name via the touchscreen. Overall, a pen based interface is significantly better for our application than a keyboard/touch-pad/mouse system. Tasks such as choosing sensor streams and dragging the threshold rules are easily accomplished, without needing to use both hands.

3.5 EDITY: Compound Rules: Combining Rules

Compound Rules refers to event definitions that are formed by combining multiple simple/existing compound rules. One way to think about these rules is to consider them forming a Directed Acyclic Graph (DAG), with each Rule forming an node in the graph. Compound rules, may be derived from simple and other compound rules (that were defined before).

3.5.1 Design Logic

While the simple rules we discussed in Section: 3.4 provide a simple and effective mechanism to set threshold events for a sensor stream, real events of interest can be better inferred by observing multiple sensor streams, simultaneously.

As a example, consider our lab’s sensor network deployment. The deployment, (see Section: 5.4), consists of 150 ceiling-mounted motion sensors spread out on the third floor of the lab in addition to around 50 “Spinner” [35] nodes which incorporate Sound, Light, Temperature and Humidity sensors. Lets assume that the user wants to track whenever someone buys food from the vending machine. While such a scenario might, at first, look atypical, it illustrates several ideas. Also, such a scenario typifies the kind of complex actions we can track with simple, user-defined rules. Note that Lifton et al. [39] used a "Plug" [39, 42] sensor node to sense a similar phenomena. The "Plug" is a modified power strip with additional sensors and a radio that forms a sensor network. The Plug reports the (electric) current draw of appliances plugged in. Additionally, it gathers sound and vibration readings. The vending machine was plugged in a plug, and it was possible to infer when a purchase was made.

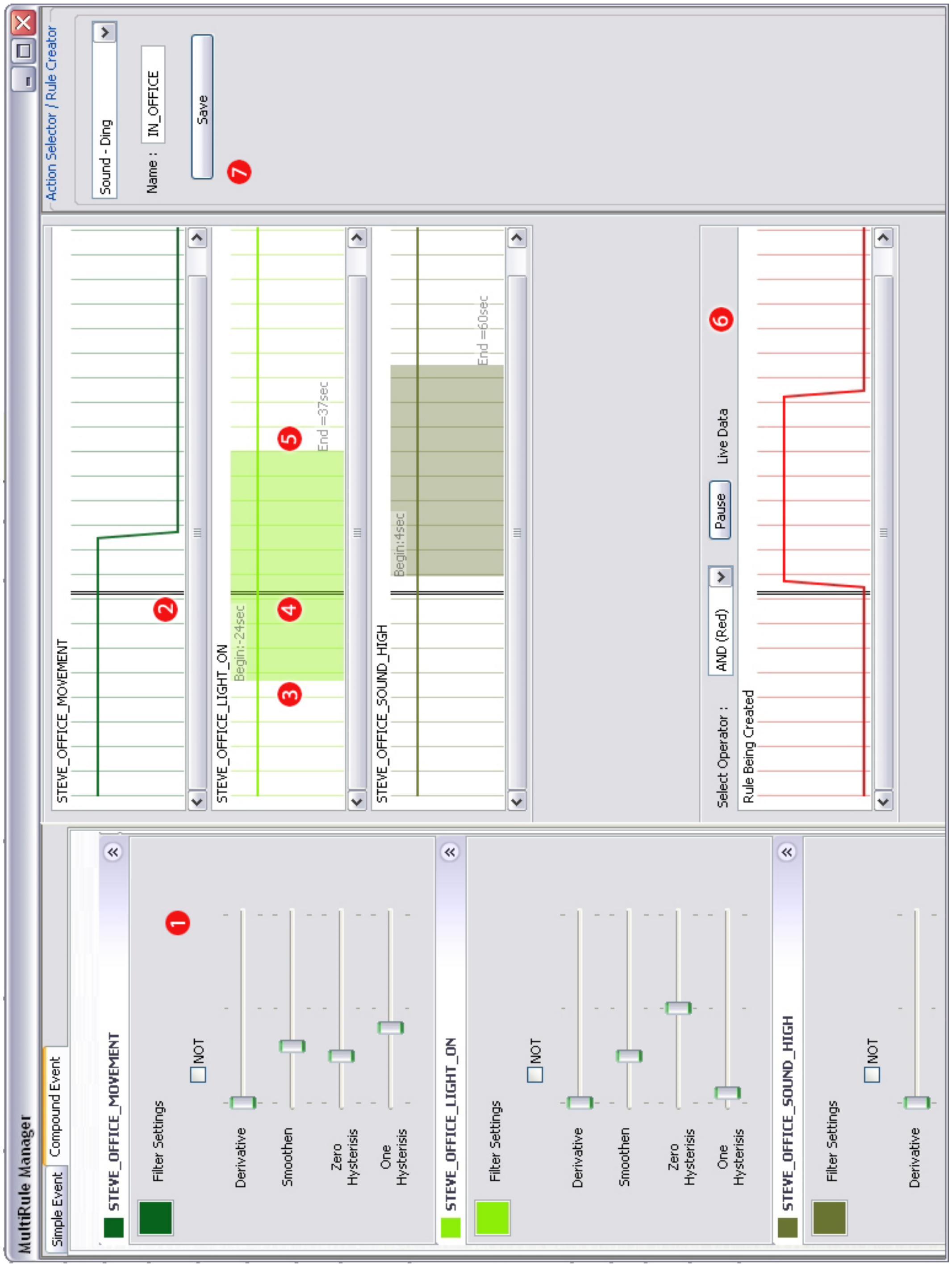
We term rules that involve multiple sensors as "Compound Rules". The constituting sensor signals can be distributed over space and, their corresponding event detection spread out over time. EDITY allows compound rules to be defined as a boolean combination of previously defined simple and compound rules. Time dependency between the rules can be described by using the concept of *time slack*, discussed below.

3.5.2 Boolean Combination of Simple Rules

A compound rule is composed of multiple simple rules combined together through a boolean operator. A compound rule then has a boolean domain (the range of simple rules and compound rules) and a boolean range.

In the UI (Figure: 3.5.2), clicking on the Compound Event Tab changes the view to the compound panel. The left pane pane now shows a list of existing simple and compound rules.

The term *component rules* refers to existing simple and compound rules that participate in formation of a new compound rule. The sum total of existing simple/-compound rules form the superset of component rules.



Select Operator : AND (Red)

Sound - Ding

Name : IN_OFFICE

Figure 3-11: Screenshot of the Compound Rule Interface. The left panel (1) has a list of existing rules, 3 of which are selected and have their filters set. The middle panel displays strip charts for the rules (in order of selection). The time dependency is defined by the black double line(2,4), and by the highlighted time slack in the strip charts(3,5). The bottom middle panel (6) shows the result of the rule the rule the user is currently designing(color image)

Clicking on a rule name (in the left pane, Figure: 3.5.2 (1)) toggles the inclusion of the rule as a component rule. Selecting a rule for inclusion also brings up a filter panel. A set of five standard filters (Not/Invert, Smooth, Derivative, De-glitch, Positive and Negative Hysteresis, see Section: 3.4.2) can be applied to the component rules prior to using it in the new rule. Note that for a compound rule, more than one component rules can be selected at a time.

In the middle pane, a new strip chart is created for new rule that shows near-real-time output of the component rule. The strip charts are color coded (lines drawn with the color of component rule, the color specified in the left pane). A pair of draggable vertical lines on the strip chart indicate the permissible time slack (discussed later).

A drop down menu at the bottom of the middle pane allows selection of the boolean operator combining the component rules. The current options are "And," "Or" and "XOR". The output of the rule is displayed in a strip chart at the bottom of the middle pane (Figure: 3.5.2 (6)). Note that the signal filter pane (left pane) also contains the "Not" modifier.

Ease of use of boolean operators

End-users are known to face problems when designing database queries that involve boolean operators, the so called "Boolean bottleneck"[24, 65].

Some of the reasons behind this difficulty are (1) difficulty in the use of parenthesis and order of evaluation when specifying queries, and (2) confusing the boolean operators AND, and OR with their counterparts in common English language.

The Ubicorder's EDITY system parenthesis every component rule, i.e., it evaluates a component rule completely before plugging in its boolean value into its compound rule. It this manner, the user does not add explicit parenthesis, but instead, the parenthesis are inserted implicitly at the cost of adding additional layers of compound rules.

The EDITY system addresses the second problem, i.e., confusion between logical

AND/OR with normal English usage of “and”, and “or”. First, we argue that our model of AND/OR operators closely parallels their equivalent English usage. In our context, such terms define linkages among the truth stage of component rules. Each component rule itself might map to some observable event. For example, a compound rule to detect if someone is in the office might consist of a component rule for increased light level, and another one for increased sound level. In this case, the condition will be defined in usual English as “”A person is in the office if the light is switched on *and* the sound level is high”. Secondly, by displaying to the user the graphical output of the rule while she devises it, we encourage the user to experiment rather than analyze; the interface lets the user quickly see the result of the rule she has so far created. Therefore, the user mixes-and-matches the output of the rule to the desired output.

Finally, we would like to state that the target audience of the Ubicorder is people with some experience in the sciences and engineering. Boolean operators are sufficiently common that most people with such a background understand it. In our user study, only one subject reported having no familiarity with boolean operators.

3.5.3 Time Dependency

One of the challenges of defining compound rules is to describe the temporal connection between the different component rule events that define a compound rule.

EDITY supports specification of time dependency in the form of *Event Y happens between (p,q) seconds after Event X*. That is, event Y happened at least p seconds after X AND that event Y happened at most q seconds after X. Note that p and q can be negative numbers.

A concept of *time slack* is introduced. A time slack means that a component rule, if it is *ever* true within a specified time window, will be construed as being true. The beginning and end of the time slack window is specified with respect to the first rule. The first rule, by definition then, has no time slack (instead, it sets the zero point).

Graphically, the time slack is set by dragging two vertical lines on the strip chart. The zero line is specified in the first rule, i.e., the rule first selected and therefore displayed at the top of the strip chart pane (Figure: 3.5.2 (2) shows the line). The zero line is then synchronized across all the other strip charts (for example, Figure 3.5.2 (4) shows one such line). Two additional time slack lines are also drawn in all but the first strip chart, corresponding to time slack for each component rule.

At this point, three lines are visible on a strip chart (except on the strip chart corresponding to the first selected sensor, in which, only the "zero point" line is visible) . The first line (Figure: 3.5.2 (4)) indicates the "zero" point, as set in the first component rule, and is drawn as a thin black double line. The second line, (Figure: 3.5.2 (3)) drawn in the color of the current strip chart, indicates the lower bound of the time slack, i.e., the event must happen at least p seconds after the first component is found to be true. The third line (Figure: 3.5.2 (5)) specifies the maximum time slack, that is, the event must happen by the end of this time period.

Graphically, drawing the time slack highlights the region on the graph.

Also, upon entering the strip chart, the user's cursor changes to a " + " sign to indicate to the user that the region can be marked.

3.5.4 Linking Actions, Recording Compound Rules

A compound rule can be saved and recursively used as a component rule for future compound rules. Action, such as issuing keystrokes and audio playback may also be programmed.

Graphically, the right panel of the interface (Figure: 3.5.2 (7)) allows for such rule setting. This panel is same as that discussed in the simple rule interface.

3.5.5 Advantages of Rule Setting

Parallels the human cognitive model

A compound rule roughly corresponds to the cognitive model of the way humans infer. If we hear the clanging of pots and the cooking range is on - someone is probably cooking. If we hear the clanging of pots and the water running, someone is probably scrubbing the dishes.

Modularity

The modular approach ensures that simpler rules can be defined and debugged completely before more complex rules are defined. Additionally, this modularity lends itself well to sharing of rules. A repository of rules describing standard states of a given space can be incorporated.

The idea of building compound rules from simple rules is similar to the idea of *subsumption architecture* [11] in the field of Artificial Intelligence. In subsumption architecture, complicated intelligent behavior is decomposed into many “simple” behavioral modules. Each module implements a particular goal of the agent. Each goal forms a layer, and subsumes the underlying layers, and each layer is tested and debugged individually. Once a layer works well, higher order layers can be built.

For example, a robot’s lowest layer could be – Avoid colliding into a wall. On top of this, a layer incorporating the robot’s objective to walk around can subsume the lower layer, i.e., wander around but ensure no collision happens.

Finally, this approach also allows a subset of basic rules to be designed by domain experts (a scenario similar to today’s appliance repair man, but here the technician is familiar with the location and physiology of the sensor data streams). The non-expert end-users can then construct more complex compound rules, building upon these expert designed simple rules.

3.6 Sample Walkthrough

This section describes, by example, how a set of simple and compound rules are created.

The task is to design a rule to have the Ubicorder count the number of people who purchase food from the a vending machine. The vending machine is located in the kitchen area, on the third floor of the MIT Media Lab. We have at our disposal, ceiling-mounted movement sensors, and a “Spinner” [35] sensor node (with a microphone) mounted on the vending machine. The Spinner node also has a IR LED that continually transmits a signature.

One of the many possible rule set that can be used is :

- Single movement sensor triggered (simple-rule A). Multiple instances of this type of rule, one for each movement sensor involved. Total of five motion sensors used. (Simple Rule A,B,C,D,E)

- Multiple movement sensors triggered (Time skewed combination of simple rule A,B,C,D,E), indicative of someone walking by (compound-rule (i))

- Sound Level on the Spinner Node registering a “high” value (simple-rule F)

- Compound-rule combining compound-rule (i) and simple-rule F (compound-rule (ii)).

The first step is to locate the vending machine. The Ubicorder’s map has an area labeled as “kitchen,” a likely place for the vending machine. As the user walks toward the vending machine, the user’s approximate location is displayed on the Ubicorder. The user also sees the motion sensor icons “pop out,” indicating that someone is walking underneath them. This leads to the first step, i.e., of defining a rule corresponding to someone walking underneath a motion sensor. Pressing the “Create Event” button in the browser brings up the rule definition and evaluation system (EDITY).

3.6.1 Simple-Rule A,B,C,D,E

Moving underneath the sensor (performing the action) and looking at the sensor signal at the same time, the pulse corresponding to the motion is clearly visible. Set the lower threshold greater than zero, and the upper threshold at any position above binary one. Save this rule as “Movement Rule A”. Set the action to “None,” and save the rule.

Create similar rules for the next four movement sensors.

3.6.2 Compound Rule (i)

Clicking on the compound rule tab brings up the compound rule definition and evaluation panel. In the left pane are a list of simple rules that have been created until now, i.e., Movement Rule A, B,C,D,E. Clicking on the rule in the left pane shows the (real-time) output of the rule in the middle pane. Each rule is displayed in a separate graph, stacked one after the other.

The rules should be selected (in the left panel) in the order that the person walks though. That is, rule A, B, C, D, E. Corresponding strip charts appear in the same order on the right side.

Now as the user walks across, multiple (simple) events (A,B,C,D,E) are seen in the pane. The user “Pauses” the data streams (By clicking on the pause button), Selects the “And” boolean operator. The other choices available are “Or” and “Xor”.

3.6.3 Time Slack

Notice the black vertical line across the strip charts. The line is synchronized across the strip charts and indicates where in the strip charts the event occurred. The line in the strip chart placed first (corresponding to the sensor rule first selected in the left pane) corresponds to the “Zero Point”. The user then highlights areas in other graphs, indicative of the time slack range. For example, for the second graph, the vertical bar is dragged to the right by a time scale that corresponds to two seconds. In that case, if the second rule fires within time [0,2] second of the first one, then it is considered true. For the third strip chart, there are 2 vertical lines - color coded according to the rule they denote. For this strip chart, the user can denote a time relative to the others.

3.6.4 Result

The result of this compound rule is displayed, in real-time, in the display pane at the bottom of the screen. The result is also a boolean (true/false) data stream. Save this rule as “Walking to Vending Machine”.

3.6.5 Simple-Rule F

We next design a simple-rule corresponding to the sound level for a typical vending machine operation. Given the location of the spinner node, the sound emitted when a purchased product falls into the vending bin (“thud”) causes a brief spike in the sound level.

The user is physically standing in front of the vending machine and observes a product fall and the corresponding sound level.

3.6.6 Compound Rule (ii)

Finally, we want to combine the knowledge that when a person is purchasing something from a vending machine, it means that someone is walking up to it, following which, something drops in the vending bin.

In the Compound Rule (left pane), a list of available rules (both simple and previously created compound rules) is displayed.

The Compound Rule (ii) is triggered as soon as all of its conditions are met. As soon as the rule triggers, a pulse is displayed. Following that, the rule corresponding to the sound level (Simple-Rule F) is triggered.

After setting the rules and the corresponding time slacks, the "action" here is to emit an operating system "+" keystroke. The action pane contains a Java Robots [29] hook, which allows any particular action to be saved. A separate program increments a counter every time the "+" keystroke is received.

3.6.7 Overall Discussion

Notice that this rule could not have been accomplished simply by using simple rule F and compound rule (i). Consider the case when a person makes two purchases. A sound-only trigger will be unable to differentiate among two people making a single purchase each from one person making two purchases. Similarly, the walking-only rule will not be able to differentiate a purchase or a person walking by. Being able to combine multiple sensors lowers the false positives.

Overall, combining multiple sensors (Sensor Fusion) provides a more expressive language and improves recognition accuracy. Recursively building rules brings the advantages of modularity to the rule making process. The use of general purpose, "high-level" sensors goes well with the idea of using them as components for other rules; "high-level" sensor outputs can be re-used as components for several compound rules. Finally, the process of building these rules enables end users to define and extract meaningful information from sensor data.

Chapter 4

Middleware for Inference and Visualization

4.1 Overview

Middleware make it easy to *plug into* services provided by existing infrastructure. The Ubicorder/EDITY system can be used as a middleware for the development of end-user facing front-ends that visualize and infer sensor data gleaned.

Significant effort has gone into middleware that caters to the designers and programmers of sensor networks. For example, Tiny DB [45] is a example of a of a query processing middleware for sensor networks, Levis et al.'s Mate [37] is a virtual machine for sensor networks to abstract away the complexity of the underlying sensor network hardware and software. Molla et al. [52] provide a survey of popular sensor network middleware platforms.

The Ubicorder/EDITY system addresses the need for easy-to-build interface to expose sensor data. Typically, application programmers (as opposed to sensor-network programmers) with limited knowledge of the sensor network will be able to design front-ends to expose the facilities of the sensor network to the user. Interfaces that do *one task well* can be designed, hiding all the complexity of the underlying sensor network, perhaps even the presence of such a network.

Such a front-end is relatively easy to code up by using the Ubicorder/EDITY system as a middleware. The front-end can be developed in a graphical language such as Adobe Flash [6] /ActionScript [5], with the mathematical heavy lifting accomplished by the Ubicorder/EDITY back-end.

4.2 Example Interface

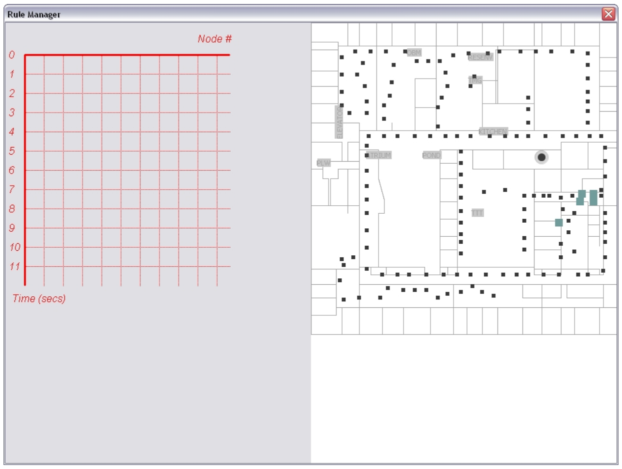


Figure 4-1: Step 0 : A floor plan with the location of relevant sensor nodes. Note that only the MERL nodes are used for this task, and so other type of nodes are not displayed.

We have built one such interface. As a precursor to the EDITY system, we built an interface to detect people walk. Later, upon completion of EDITY, we discarded the rule matching and sensor parsing components of the old interface and used it only as a front-end, with EDITY doing the remaining tasks.

This particular interface was designed to let end users define events corresponding to people walking from place A to place B, indoors, and observed by the ceiling-mounted movement sensors. In our interface, the user marks the path of interest by 'drawing' on a floor plan, indicative of the path of interest. The said system then 'designs' the rules, and feeds it into the Ubicorder/EDITY subsystem. Further, a simple, “tweaking” interface is also provided to allow the user to adjust for extended delays between two movement nodes being triggered (when, for example, the movement sensors are positioned at unequally spaced intervals).

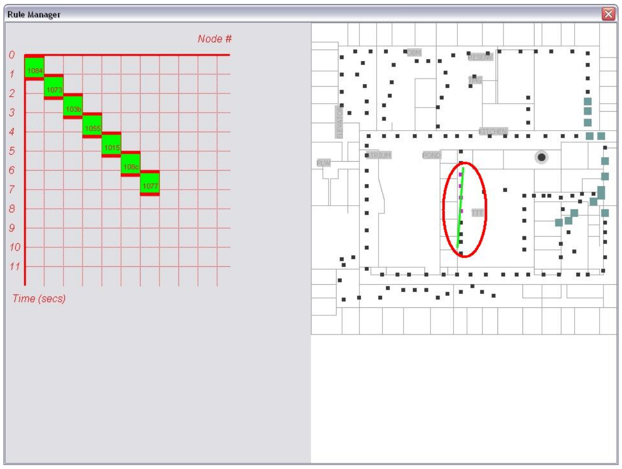


Figure 4-2: Marking the path (green line, circled for clarity in the figure above) that the user wants to detect people walking on. The left pane contains manipulable time data.

Figure: 4-1 shows the “rest state” prior to the definition of any rule. The right panel shows a map of the area, overlaid with icons denoting motion sensors. The left panel shows a time chart graph for tweaking rules.

A rule can be drawn by merely dragging the pen to indicate the path of interest. This is shown in Figure: 4-2. Once the path is drawn, the corresponding nodes light up (change color to magenta), one after the other (to indicate the direction of the path), see Figure: 4-3. The left panel shows a time chart graph for tweaking rules. The X-axis of this graph is labeled for each node ID, the Y-axis indicates time. The boxes on the left half of the display, denote the *time slack*, the time relationship between firing of the movement sensors.

The time slack can be tweaked by dragging the lines to sculpt the rule and to change the time slack. This is shown in Figure: 4-4, with the second time slack being modified to overlap with the third.

This ties back to the scenario of the “repair man” defining the basic rules. Instead

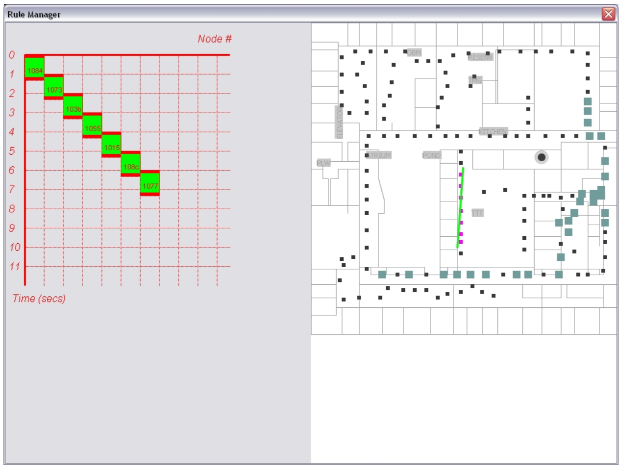


Figure 4-3: Nodes changing color in the order of the drawn path, indicates the direction of walking that will be detected, and the time between which adjacent nodes must detect motion.

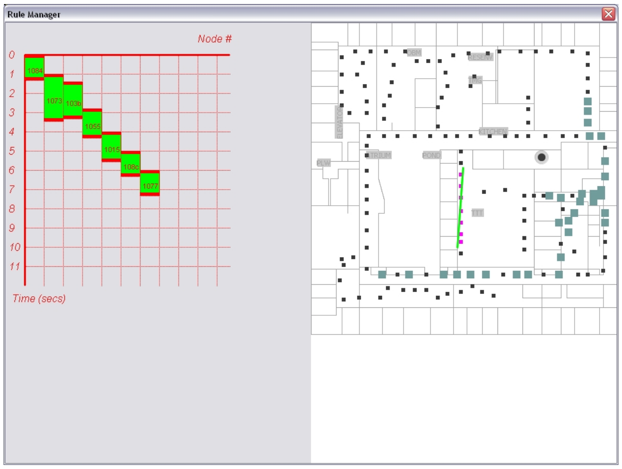


Figure 4-4: Manipulating the time data in the left pane to adjust *time slack.*

of a “repair-man” designing basic rules to be built upon, they can provide the user with a easy-to-use front-end interface, designed specifically for a single task. In the above example, the task is to detect people walking from point A to point B. Other scenarios could be, for example, a “presence” detector.

Chapter 5

Software And Hardware Implementation

This chapter describes the hardware and software components of the Ubicorder, and provides the design rationale for those choices.

5.1 Summary and Goals

The goal of this work are to enable users to browse information derived from heterogeneous sensor networks and to infer actions from received sensor data. In order to accomplish this higher level objective, a number of more specific goals emerged. These specific objectives included:

- Acquisition of sensor data from sensor nodes, and presenting a homogeneous interface of such sensor data to the rest of the system.

- A portable platform for display and interaction with sensor data that incorporates a compass, wireless radio and an IR receiver to support situated user navigation and exploration of their physical surroundings.

- A graphical interface for *in-situ* browsing and visualization of sensor data.

- An interactive graphical scripting environment for defining and manipulating higher-level events.

The following chapter explains the development of software and hardware subsystems, and justifies the design decisions made to achieve these objectives.

5.2 High Level Description of the System

The current Ubicorder comprises of a touchscreen (pen sensitive) Tablet PC with Wi-Fi, and a Universal Serial Bus (USB) port. It is augmented with a custom built hardware board that incorporates a three-axis tilt-compensated compass, an infrared receiver/transmitter, and a wireless radio. The custom hardware talks to the Tablet PC over a USB connection. Firmware was written for the extra hardware, and a driver stub was written the for Tablet PC to be able to talk to the given device over USB. The following section describes in detail the hardware components of the system.

5.3 Hardware Overview

This section describes the hardware used and built for this project.

5.3.1 Mobile Computer System

A mobile computer system provides the base functionality for the platform. The system was selected based on its portability, processing and networking capabilities, ease of augmenting/attaching additional hardware, and ease of software development.

It was essential that the system be portable, and that it allow the user to simultaneously interact with the real world and with the sensor stream describing the real world in real-time. As discussed later, this enabled the user to iteratively create and experiment with defining higher-level events occurring at various places.

The system also features a touchscreen. Pen based interactions are well suited in scenarios where the user needs to select icons and drag lines. Touchscreen enabled Tablet PC's are often used for such tasks; for example, in [62], the authors describe LeafView, a tablet-based user interface for an electronic botanical field guide.

Based on these parameters, the Lenovo ThinkPad X61 'ultralight' tablet PC (X61) emerged as the best choice for our initial implementation. The X61 has a comfortable form factor (3.5 lbs, 12.1" multi touch Display), incorporates a fast 1.8 G Hz dual core processor, comes with Wi-Fi and an USB port. It runs Microsoft Windows XP Tablet Version, which supports the same system calls as a standard Windows XP machine. The processor is fast enough for our visualization tasks.

An important trade-off to consider was size and portability. Smaller form-factor devices such as a personal digital assistant (PDA) is more convenient for carrying around. However, for our application, a large screen size is useful in order to display the complete scenario being authored, i.e., show the selected sensor, the rule being designed for it, and the corresponding action. Further, small form factor devices have limited processing power. Processing capabilities are required because the application of rules to sensor data currently happens on the handheld. In future implementations, such a task could be off-loaded to a database running on a server. Future versions of the Ubicorder will be designed to run on such devices.

5.3.2 Extra Hardware

Displaying the user's location and orientation relative to other sensor nodes on the screen enables easy navigation and provides additional context information to situate the user with respect to their environment.

To realize this functionality, we augmented our base computer with a Zigbee [7] radio, an infrared (IR) transmitter-receiver and a digital compass. In particular, circuits and devices designed by fellow Responsive Environments Group Research Assistant Mathew Laibowitz, for his "Spinner" sensor network [35], were appropriated and adapted for these purposes.

Digital Compass

The digital compass required for this purpose must be tilt compensated, so as to allow the user to hold the device at a convenient angle. Further, the compass must compensate for extraneous magnetic fields and nearby ferrous material, not uncommon inside the building. For this purpose, the Honeywell HMC6343S, a 3-axis, tilt and hard-iron compensated compass embedded on the "Spinner" [35] node, works well. The compass talks to the microcontroller over the I^2C protocol [56], and provides the Ubicorder's heading, yaw and roll values.

Infrared Receiver

The infrared port is used to realize a 'point and select' affordance, i.e., the user can point toward a sensor node and select it on screen. A subset of sensor nodes (i.e., Laibowitz's "Spinner" sensor nodes) periodically emit an IR signature. By receiving this signature, the corresponding node can be selected. Further, using this information, compass data and a static look-up-table, the user's rough location can be determined.

Microcontroller

The processor used on the wearable "Spinner" node, Atmel UC3B256 [8], is a 32 bit microcontroller. Accordingly, it was used as a bridge between the sensors and the X61 tablet. The microcontroller supports USB, has multiple Analog to Digital(ADC) ports, and supports I^2C [56] protocol for communicating with the compass. A simple instruction set was defined for communication between the microcontroller and the PC.

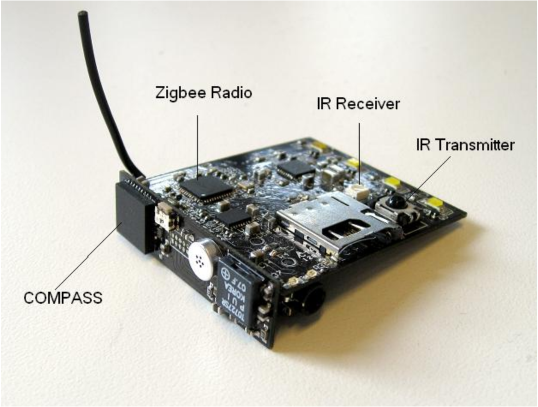


Figure 5-1: Additional Hardware

Zigbee Radio

The device uses a CC2431 Zigbee radio [27]. This Zigbee Radio is also used on the "Shirt-Lapel Pin" sensor nodes in the Spinner network [35], can communicate with other nodes, and determine the radio signal strength (RSSI). The latter information is used to coarsely estimate the user's location with respect to the static nodes of the sensor network.

Also note that this radio is set to a "peer" profile, i.e., it joins an already existing Zigbee [7] network. Although these Zigbee radios could be used for directly acquiring data from the sensor nodes, we did not use it for this purpose, since all nodes of our sensor network push data to a network socket, which we connect to over the Wi-Fi network. The current Ubicorder only uses the Zigbee network for RSSI-based localization relative to an array of fixed reference nodes.

USB

A USB connection is used to communicate between the X61 Tablet and the microcontroller. The USB connection also provides a 5V DC power output, which is used to power the additional hardware.

The microcontroller was programmed so that it appears as a Connected Device Configuration (CDC) slave, with the X61 tablet becoming the master. Although

Windows has native support for CDC devices, it was essential to write a .inf file to provide information about the device and files which were to be installed by the system. The USB connection then appears as a virtual com port, via which serial commands and data can be exchanged.

5.4 Sensor Networks

There are two available sensor networks for this implementation of the Ubicorder. These were the “Spinner” [35] and the MERL [31] networks. The ideas behind Ubicorder are not directly tied to the sensor network it communicates with.

5.4.1 MERL Sensor Network

The MERL sensor network consists of 150 ceiling-mounted sensor nodes deployed on the third floor of the MIT Media Lab. The nodes (Figure: 5.4.1) and the networking infrastructure was designed at the Mitsubishi Electric Research Labs (MERL), Cambridge, MA. These devices were originally installed at the MIT Media Lab by Ishwinder Kaur for collecting and analyzing data related to space usage patterns (discussed in Section: 2.2 and [31]).

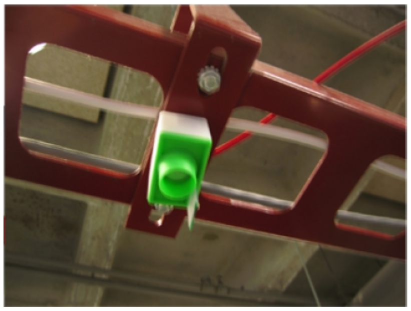


Figure 5-2: A “MERL” [31] movement node mounted on the ceiling

The MERL network reports boolean pings when the nodes detect motion underneath. Each node wirelessly communicates data to one of the seven Zigbee-Ethernet gateway devices spread around the deployment area. These gateways, in turn, make the MERL data available over the Ethernet/Wi-Fi network. The Ubicorder pulls the data over the Wi-Fi network by directly connecting to each of the seven gateways.

5.4.2 Spinner Sensor Network

The "Spinner" network [35] has been designed by Mathew Laibowitz of the MIT Media Lab's Responsive Environments research group (Resenv) as a part of the PhD research. The network is originally designed to identify, detect and record human behavior with an intent to put together a cohesive narrative that conveys a larger overall meaning.

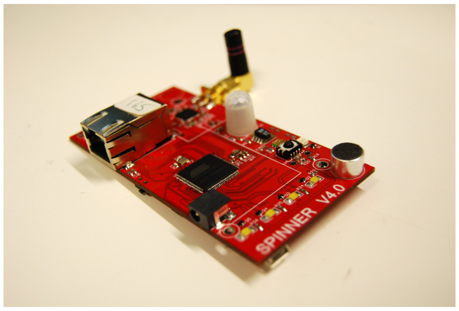


Figure 5-3: A "Spinner" sensor node, designed by Mathew Laibowitz

In particular, the sensor nodes incorporate multiple sensors: temperature, light, stereo microphones, movement and vibration. Additionally, some (future) nodes will also incorporate a video camera. There is also a wearable component of the node planned. A 50 node network is currently being rolled out in the Media Lab. The final network topology follows a publisher-subscriber model, with all the nodes pushing data to a central data repository over a wired data connection. However, in the current deployment, the nodes are connected to the Media Lab's (ML) Ethernet network, and the Ubicorder directly connects to these nodes (by making a Wi-Fi connection to the ML's network).

5.5 Software

The bulk of the code was written in Java 6.0 programming language. In its present form, the code spans a total of around 6K lines.

In addition to the Java code, the author also wrote a subset of the microcontroller firmware (in the C programming language). In particular, the code dealt with acquiring compass and infrared signature data and pushing it to USB. Appendix A and B lists the Java 6 and microcontroller firmware respectively.

5.5.1 Software Environment

The initial look and feel prototype was implemented to run on Adobe Flash CS3 professional [6] and used ActionScript 3 (AS3) [5] scripting language. Flash/AS3 is focused on allowing interaction designers to iterate quickly over multiple UI approaches; it has a wide selection of built-in primitives for User Interface (UI) design. As an analogy, many consider Flash AS3 as the Matlab (i.e., quick and easy but computationally expensive) of the UI design world. We found, however, that the Flash/AS3 setup was unable to handle large sensor data streams. Further, the platform did not support simultaneous multiple thread execution. After making an initial prototype in Flash/AS3, we abandoned it for a Java implementation.

Java provides a high performance, threading-friendly, and robust setup for complete implementation. Multiple thread support was essential for us; our requirements included parsing and incorporating data from multiple sensor network gateways, while, at the same time, providing a responsive UI. Unlike Flash/AS3, Java supported multiple-thread execution.

The Software Widgets Toolkit (SWT) was chosen as the UI library for our implementation. The choice was between SWT and the Swing framework. The SWT framework uses native (operating system) window components, and gives better performance. Additionally, the initial design goals of the project included porting this

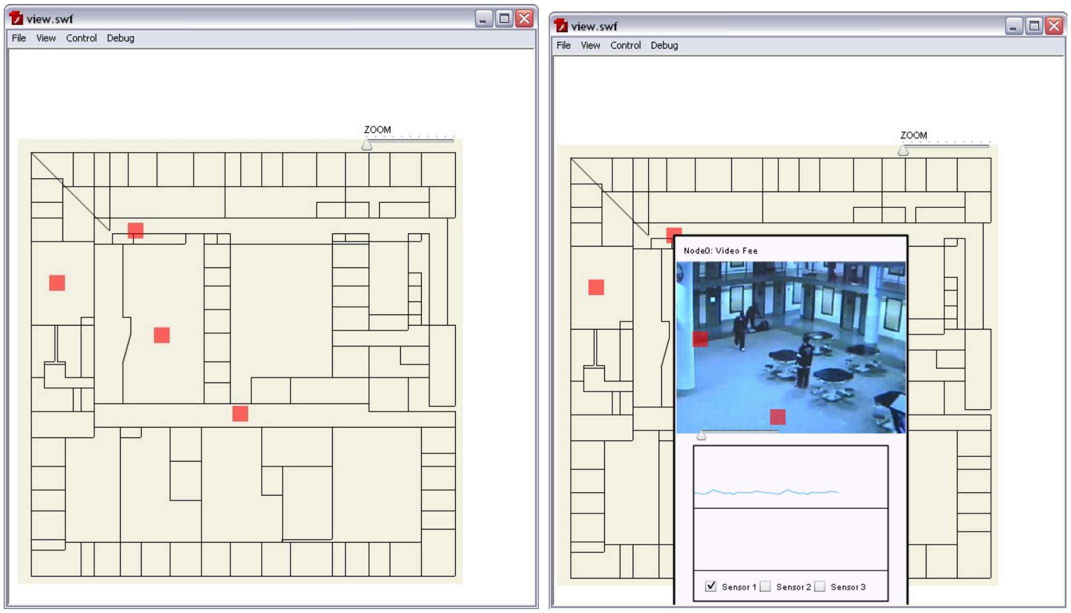


Figure 5-4: Early Flash Prototypes. Double Clicking on a "Node" brings up raw data and a video feed, where available

framework onto the Nokia N810 Internet Tablet [55], a 128mm x 72 mm sized handheld. At the time of writing, the N810's Java port only supported SWT and not Swing.

Sensor Network Data Acquisition

Basic calibration information, such as location/type of static sensor nodes, format of sensor network data, and network addresses of the gateways are currently hard-coded into the main program.

Sensor data is pulled directly from the sensor node gateways, which, in our case, are connected to a wired Ethernet network. These gateways also listen (bind) for incoming socket connections, and can push data to these connections. The X61 tablet finally pulls this data over Wi-Fi.

Since we used heterogeneous sensor networks, with each exposing data in its own format, a list of parsers were written that would present a uniform view of the sensor data to the rest of Ubicorder's software.

Further, there were several quirks in managing different types of sensor gateways. For example, our motion sensor network only reported movement pings, i.e., only reports data when the motion sensor node observed motion. In contrast, other sensors, such as light/sound sensors (part of the "Spinner" network) exposed complete analog time-series data. For the rule-setting layer of the system (EDITY: see Section: 3.2), a uniform sensor interface is useful. Motion sensor data can be converted into time-series data. However, to do that for every motion sensor node, and at all times, would be wasteful (Over 150 such nodes are presently deployed). We worked around this by generating motion sensor data as a time-series when and if such a node is monitored as per a user-defined rule. In particular, a startStreaming() function is called when a rule starts monitoring a sensor, and a stopStreaming() function is called when the rule is disposed. Every sensor exposes these functions (defined in the "SensorInterface" interface).

The Ubicorder makes direct (Wi-Fi, 802.11 g/b) socket connections to each of these gateways and retrieves near-real-time sensor data. While this approach does not scale well, it was acceptable for our usage scenarios. Ideally, the sensor data from the gateway should be pushed to a replicated, query-able database to which the Ubicorder can talk.

In particular, a static list of IP addresses and gateways is maintained, along with the type of sensor data this gateway collects. A software thread corresponding to each gateway is tied to a custom parser corresponding to the type of data pushed by the gateway.

5.5.2 Discussion

Latency was one of the key constraints to be managed. For the creation of rules (EDITY system, discussed in Section: 3.2), we rely on the user being able to easily correlate physically observed phenomena with the sensor data. In order to minimize latency, a notify/observe protocol was used to communicate among multiple threads.

It is worth mentioning that we began by strictly adhering to the Model View Controller [19] design pattern. However, during the course of our implementation, we found following a rigid pattern to be cumbersome and unnecessary. One reason for this was because only one programmer (present author) designed, implemented, tested and debugged the entire code base. Also, given that this was a research project, new ideas emerged while others were being implemented. Hence, there was never a freeze on the features and the design.

It is difficult to describe the code itself. In the following tables, we list and explain the various threads that execute in the program. Next we list all the classes and interfaces used.

The software itself is organized into several classes. Note that in the Extends/Implements Interface panel, classes written by the author are *italicized*. Other classes are provided by the Java.util and Java.eclipse.swt.* classes.

Table 5.1: Threads: Functionality, and Lifetime

Threads, Functionality and Lifetime			
No.	**Thread Name:**	**Description:**	**Lifetime:**
1	MerlListener	Connects to Each MERL Sensor Network Gateway, Parses the Merl Data and pushes it to corresponding nodes. Also notifies registered listeners of new data for a particular node.	Lifetime of the Main Program
2	SpinWerk-Listener	Connects to Each Spinner Sensor Network Gateway, Parses the Spinner Data and pushes it to corresponding nodes	Lifetime of the Main Program
3	ComPortFuncs	Talks over the USB to additional Hardware(Compass, Radio, IR), Gets the Compass Data and the nearest Static Spinner Node ID.	Lifetime of the Main Program
4	BigMapView	Main Browsing UI, Displays the floor plan, overlays the user's present location and orientation, draws the sensor network node.	life-time of the Main Program
5	MultiRuleView	Event Definition / Rule Creation User Interface.	Active when User Clicks the "Create Rule" Button, until user closes the Rule Creation/Modification Window
6	Simple Rule Threads	Check if a Simple Rule satisfied	Created when corresponding sensor object notifies() of new data, destroyed after check is complete
7	Compound Rule Threads	Check if a Compound Rule is satisfied	Created when corresponding Simple Rules(s) notify of new Data.

Table 5.2: Class Structure of Ubicorder's Software

Class Structure of Ubicorder's Software				
No.	**Class Name:**	**Description:**	**Extends:**	**Implements Interface:**
1	Sensor	One instance for each Sensor.	Observable	*SensorInterface*
2	Node	One instance for each Node. A node may have multiple sensors	Observable	Listener
3	SimpleRule	One instance for each simple rule created	Observable	*SensorInterface*, Observer
4	CompoundRule	One instance for each compound rule	*SimpleRule*, Observable	*SensorInterface*, Observer
5	Filter	One instance for each "Type" of filter		
6	SmoothFilter	Smoothening Filter, One instance for every sensor stream displayed at the time	*Filter*	*FilterInterface*
7	Differentiate Filter	Differentiating Filter, One instance for every sensor stream displayed at the time	*Filter*	*FilterInterface*
8	MerlListener	One instance for each gateway of the Merl Sensors.		Runnable
9	SpinWerkListener	An instance for each Spinner Node(The node is also the gateway)	*MerlListener*	Runnable
10	ComPortFuncs	One instance for each Serial Port Connection Opened.		Runnable
11	BigMapView	Main Browsing UI.		Runnable, PaintListener, Listener, SelectionListener
12	MultiRuleView	Event Definition / Rule Creation UI.		SelectionListener, Listener, PaintListener, Observer
13	Map	Instance of floor plan, overlaid with icons denoting nodes		PaintListener, Listener
14	MarkedMap	A map that lets nodes to be selected by clicking on them	*Map*	PaintListener, Listener
15	SuperExpandBar	Left Panel Bar of the Rule Editor	Observable, ExpandBar	ExpandListener, PaintListener, Listener

Table 5.3: Class Structure of Ubicorder's Software(continued)

Class Structure of Ubicorder's Software				
No.	**Class Name:**	**Description:**	**Extends:**	**Implements Interface:**
16	StripChart	One instance for each strip chart drawn	Canvas	Listener, PaintListener, Observer
17	FilterSlider	One instance for each Filter Available	Composite	
18	MerlSensor-ListParser	One instance for each text file which contains a list of Merl Sensor ids, and their locations		

Chapter 6

Experiment

6.1 Introduction

We conducted a first-use user study to evaluate the usability and the usefulness of the Ubicorder. The study aimed to test:

- If the Ubicorder successfully enabled participants to discover, explore and browse the deployed sensor network.
- If the Ubicorder aided the participants in interpreting and therefore using sensor data.
- If designing events and rules was useful in aiding the participant to interpret data.
- The ease of use of the Ubicorder, including the browsing and the EDITY interfaces.

Further, the study aimed to understand some constraints and shortcomings of the system, thereby suggesting directions for future work.

6.2 Study Setup

6.2.1 Infrastructure

The study was conducted at the MIT Media Lab's third floor. Although the Ubicorder itself is sensor network agnostic, two sensor networks were used for the purpose of this study. Section: 5.4 provides a detailed description of the sensor networks used.

The first was a 150 node MERL sensor network that consisted of ceiling-mounted nodes with motion sensors. The second network is the "Spinner" sensor network. The final network, when completely deployed, will span around 50 nodes around the Media Lab. However, for our setup, we only deployed a total of four "Spinner" nodes on the Media Lab's third floor. Figure: 6.2.1 shows the placement of the Spinner nodes. One "Spinner" node was in the area next to the elevator, another in the kitchen. One node was deployed "Resenv" area (Responsive Environments area) near the microscope/soldering station. Another node close the Resenv Area is located within the author's office and was used while demonstrating the Ubicorder during the user study. Note how the nodes in the public areas, i.e., the elevator and the kitchen nodes, have large halo's (indicative of high sound levels). The nodes deployed in the Resenv area report low sound levels.

6.2.2 Participant Profile

A total of ten participants took part in our user study. Of these, three were female and seven were male. An additional two participants served as "pilot testers" for the study, and their answers were not included in the analysis. Participants were recruited by posting fliers around the Media Lab, and in the authors dormitory.

The participant profile spanned the course of education and age levels. Of the ten participants, three were non-engineering (science) undergraduate (rising Junior) students at MIT. Five participants were graduate students in the Media Arts and Science program at our institution, and had engineering backgrounds. Finally, the

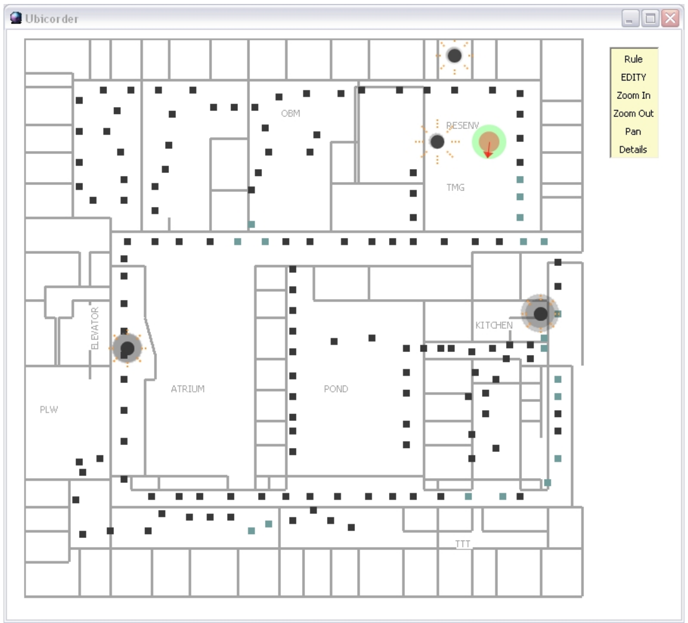
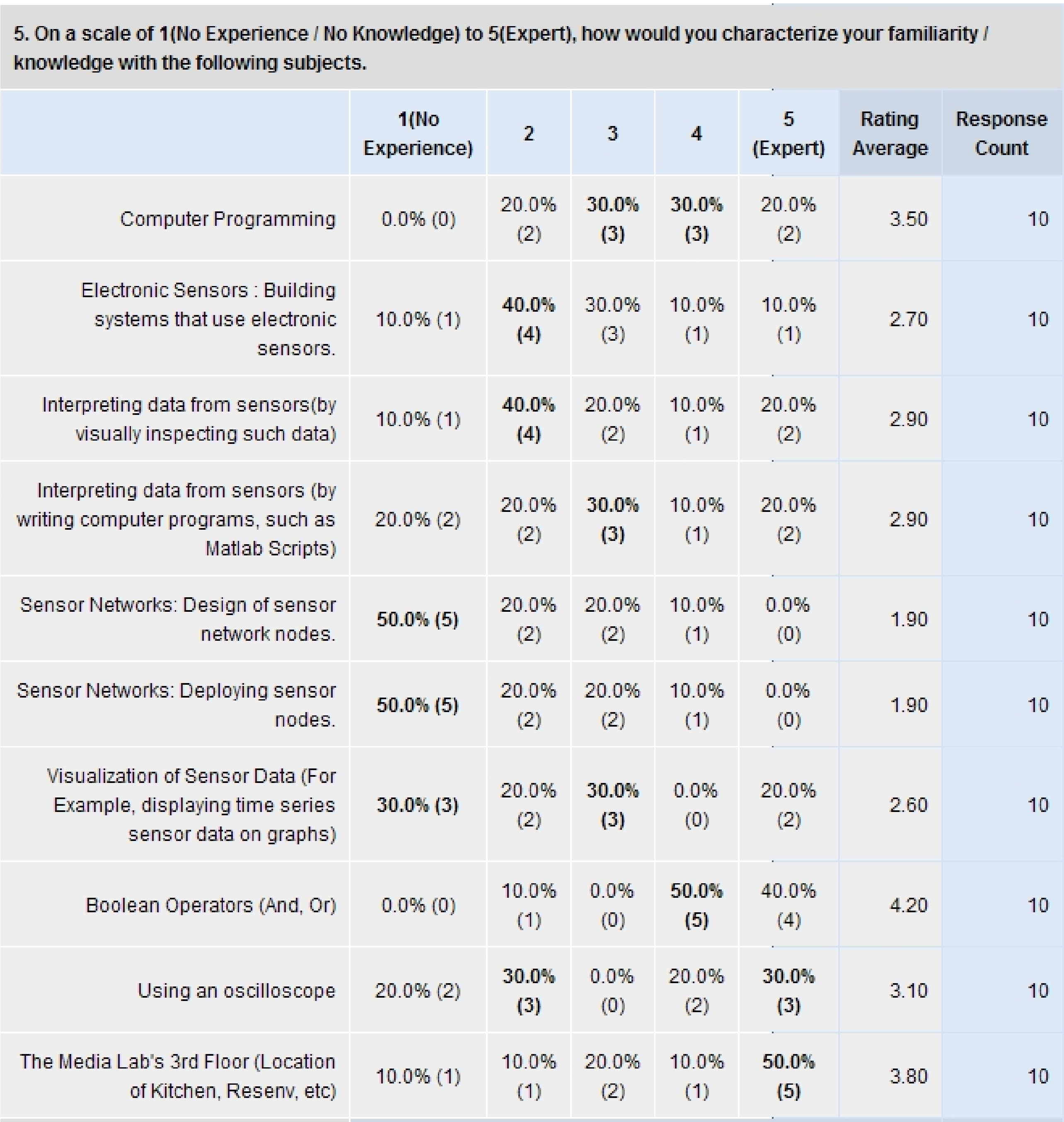


Figure 6-1: Test Setup: Location of nodes. One "Spinner" node was deployed next to the elevator, another in the kitchen. One node was deployed "Resenv" area (Responsive Environments area) near the microscope/soldering station. Another node close the Resenv Area is located within the author's office. The nodes in the public areas (elevator, kitchen), have large halo's (indicative of high sound levels). The nodes deployed in the Resenv area report low sound levels.

study also included two postdoctoral scholars, one working in the field of computer vision, and another working on holographic displays. One pilot participant was a member of our research group. Five others were students at the Media Lab.

While all participants had at least some prior programming experience, 40% of the participants had no prior knowledge of sensor networks. Half the participants were novices or had no experience in building systems that used sensors. Half the participants also had little or no prior experience in interpreting sensor data based on visual inspection (eyeballing). Most of the participants (90%) were aware of boolean operators (And, Or), and had used them before. Figures: 6-2 and 6-3 show the knowledge level of the participants.

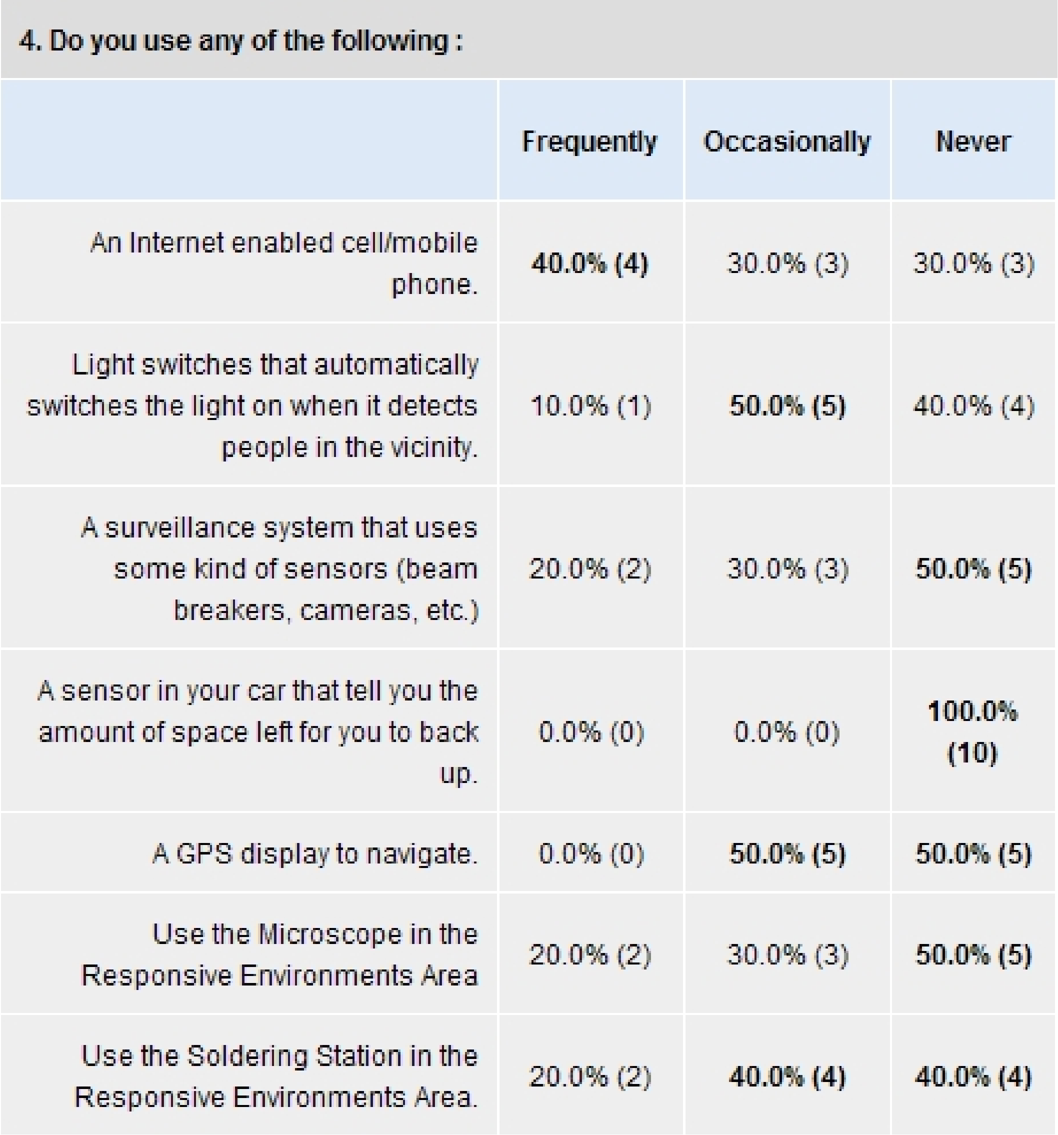
Participants were compensated with a ten dollar gift certificate for a local coffee shop.



5. On a scale of 1(No Experience / No Knowledge) to 5(Expert), how would you characterize your familiarity / knowledge with the following subjects.

	1(No Experience)	2	3	4	5 (Expert)	Rating Average	Response Count
Computer Programming	0.0% (0)	20.0% (2)	**30.0% (3)**	**30.0% (3)**	20.0% (2)	3.50	10
Electronic Sensors : Building systems that use electronic sensors.	10.0% (1)	**40.0% (4)**	30.0% (3)	10.0% (1)	10.0% (1)	2.70	10
Interpreting data from sensors(by visually inspecting such data)	10.0% (1)	**40.0% (4)**	20.0% (2)	10.0% (1)	20.0% (2)	2.90	10
Interpreting data from sensors (by writing computer programs, such as Matlab Scripts)	20.0% (2)	20.0% (2)	**30.0% (3)**	10.0% (1)	20.0% (2)	2.90	10
Sensor Networks: Design of sensor network nodes.	**50.0% (5)**	20.0% (2)	20.0% (2)	10.0% (1)	0.0% (0)	1.90	10
Sensor Networks: Deploying sensor nodes.	**50.0% (5)**	20.0% (2)	20.0% (2)	10.0% (1)	0.0% (0)	1.90	10
Visualization of Sensor Data (For Example, displaying time series sensor data on graphs)	**30.0% (3)**	20.0% (2)	**30.0% (3)**	0.0% (0)	20.0% (2)	2.60	10
Boolean Operators (And, Or)	0.0% (0)	10.0% (1)	0.0% (0)	**50.0% (5)**	40.0% (4)	4.20	10
Using an oscilloscope	20.0% (2)	**30.0% (3)**	0.0% (0)	20.0% (2)	**30.0% (3)**	3.10	10
The Media Lab's 3rd Floor (Location of Kitchen, Resenv, etc)	10.0% (1)	10.0% (1)	20.0% (2)	10.0% (1)	**50.0% (5)**	3.80	10

Figure 6-2: Participant Profile : Domain Knowledge



4. Do you use any of the following :

	Frequently	Occasionally	Never
An Internet enabled cell/mobile phone.	**40.0% (4)**	30.0% (3)	30.0% (3)
Light switches that automatically switches the light on when it detects people in the vicinity.	10.0% (1)	**50.0% (5)**	40.0% (4)
A surveillance system that uses some kind of sensors (beam breakers, cameras, etc.)	20.0% (2)	30.0% (3)	**50.0% (5)**
A sensor in your car that tell you the amount of space left for you to back up.	0.0% (0)	0.0% (0)	**100.0% (10)**
A GPS display to navigate.	0.0% (0)	**50.0% (5)**	**50.0% (5)**
Use the Microscope in the Responsive Environments Area	20.0% (2)	30.0% (3)	**50.0% (5)**
Use the Soldering Station in the Responsive Environments Area.	20.0% (2)	**40.0% (4)**	**40.0% (4)**

Figure 6-3: Participant Profile: Related Knowledge

6.3 Study Design

Each trial lasted for one hour and fifteen minutes. The test began with a pre-survey questionnaire (5 mins.), followed by a brief introduction to the system (10 mins.). The participant then performed the assigned tasks (40 mins.). This was followed by an exit survey (15 mins.), and a post study session (5 mins.) in which the interviewer answered any questions that they participant might have. The investigator (in this case, the author) logged the names of the rules created by the participants, and observed as they performed the task.

6.3.1 Profile Survey

The participants were asked to fill in a profile survey before being introduced to the system.

6.3.2 Introduction

The study started with a demonstration of the Ubicorder. We also introduced the sensor nodes available (The MERL and the Spinner Networks), and the sensing modalities each node incorporates. The participants were also instructed about using signal filters (Smoothing, Hold High, Hold Low, Not).

The participant was then asked to walk around and observe the changes on the screen. He/She could visually correlate the icons corresponding to the motion sensors triggering as he/she walked around.

6.3.3 Task

The task itself was divided into two smaller tasks.

1. *Warm Up Task*: In the initial task, the participant was handed the Ubicorder, and was asked to locate the Spinner node in a dark room (least light). He/She then physically walked to the location of the given "Spinner " node. The participant was to then program a rule such that the node's icon's color changes to red (from black) when the light is switched on. The participant was also shown the location of the light switch, and was allowed to toggle the switch. This was an example of a simple rule being designed.

Figures: 6.3.3 and 6.3.3 show one such node used for our test.

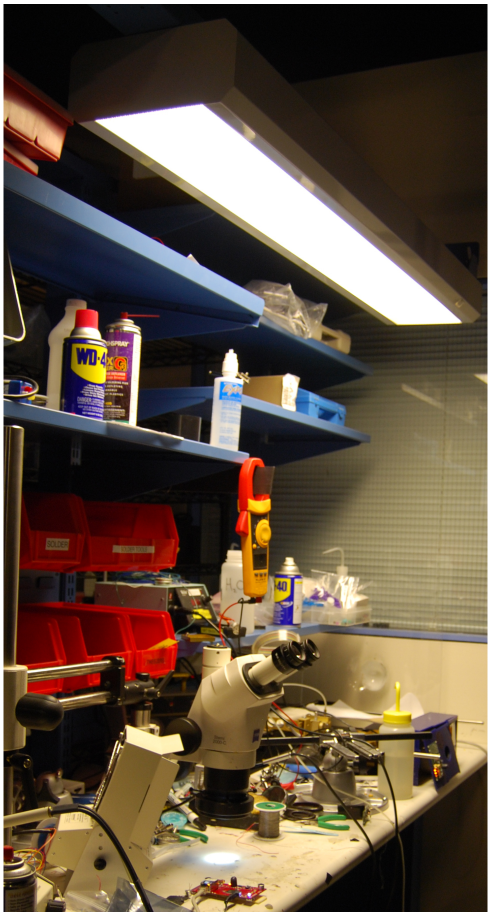


Figure 6-4: The microscope/soldering station workbench with ambient light

2. *Main Task*: The participant was taken to the location of the microscope/soldering-station workbench (Figures: 6.3.3 and 6.3.3). The participant was to design a rule to detect if the said microscope/soldering station was in use. Since several participants had never used a microscope/soldering station before, an “actor” acted out the usage scenario, i.e., the participant switches on a microscope LED spotlight, and turns on the fume exhaust (a device kept on the workbench table, containing a fan, which absorbs harmful solder fumes). The actor occasionally cleaned the soldering iron using a (moist) cleaning foam, resulting in visible water vapor emanating from the cleaning foam.

The participant was shown the LED spotlight switch, the soldering iron switch,

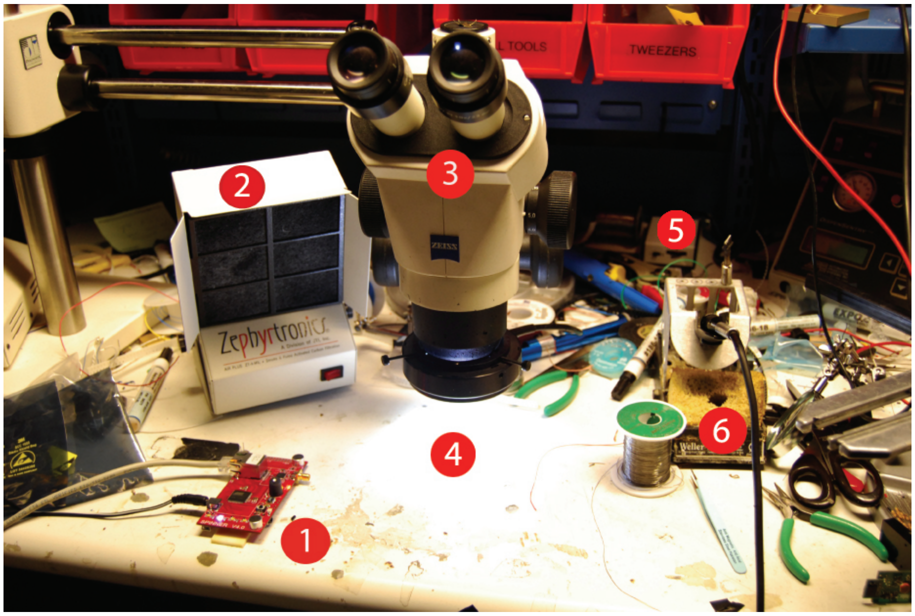


Figure 6-5: Zoomed in view of the Spinner Node deployed on the microscope/soldering workbench. (1) Spinner Node, (2) fume exhaust, a device incorporating a fan and a filter to absorb noxious solder fumes, (3) microscope, (4) Area under the microscope, illuminated by a LED spotlight, (5) LED spotlight switch, (6) Soldering Iron

and the fume exhaust switch. The participant was also shown the switch for the tube light mounted above the workbench for ambient light. Finally, the participant was informed that users may or may not switch on the ambient light when soldering, but always need to switch on the LED spotlight when soldering.

This task was chosen because it provides an opportunity for the participant to design a straightforward (but non-trivial) compound rule involving sound and light sensors. This setup also allowed participants to create more intricate rules involving additional sensors. Around half of our participants were familiar with the soldering station, and so it therefore introduced invariance with regards to prior knowledge of the task.

Finally, the participant was to program the alert or actuation to be triggered when the microscope was in use. Participants could set alerts such as audio alerts ("ding" sound) or visual alerts (box drawn around a node, changing the color of the node).

6.3.4 Exit Survey

Following the completion of the task. The participant was then asked a list of questions that evaluated the usefulness and usability of the Ubicorder. The survey also contained questions about the participants attitude toward sensor networks.

6.4 Observations and Implications

6.4.1 Completion

All participants were able to accomplish the task. Participants took a variety of approaches in defining rules, some of which had not been originally envisioned by the study-designer at the time of the design of the user study.

6.4.2 Defined Rules

For the warm up task, participants designed a rule that used light levels as the sensed modality. The participants did not face any problems.

For the main task, the participants created rules that used the light levels, and then experimented with sound, motion, temperature, and humidity. All participants used at least two modalities to detect when the microscope/soldering station was in use.

Most participants ended up devising the following set of rules:

1. Simple Rule A: Corresponding to increased light level. The light level increases when the tube light at the top of the workbench is switched on, and increases further when the LED spotlight mounted on the microscope is turned on. The participants designed a rule so that it is invariant to the user switching on the tube light. At the end, the lower threshold was set to the value corresponding to the tube light switched off and the LED spotlight switched on, while the upper threshold corresponded to both the tube light and the LED spotlight switched on.

2. Simple Rule B: Corresponding to increased sound level. When the fume exhaust fan was switched on, the increased sound level was used as a signature. Most participants also used a smoothing filter when devising a rule for the sound data.

3. Compound Rule C: Participants "Anded" the Simple Rule A and Simple Rule B, and saved the rule.

Participants also experimented with other sensor modalities. Two participants designed a simple rule that used data from the Spinner node's motion sensor. One participant used this rule in addition to the Light and Sound rule, while the other used it in place of the Simple Rule B (sound).

One participant did not design a single rule for the light, instead choosing to design two simple rules:

1. Simple Rule A1: True when light level = light level with both the tube-light and LED spotlight switched on.

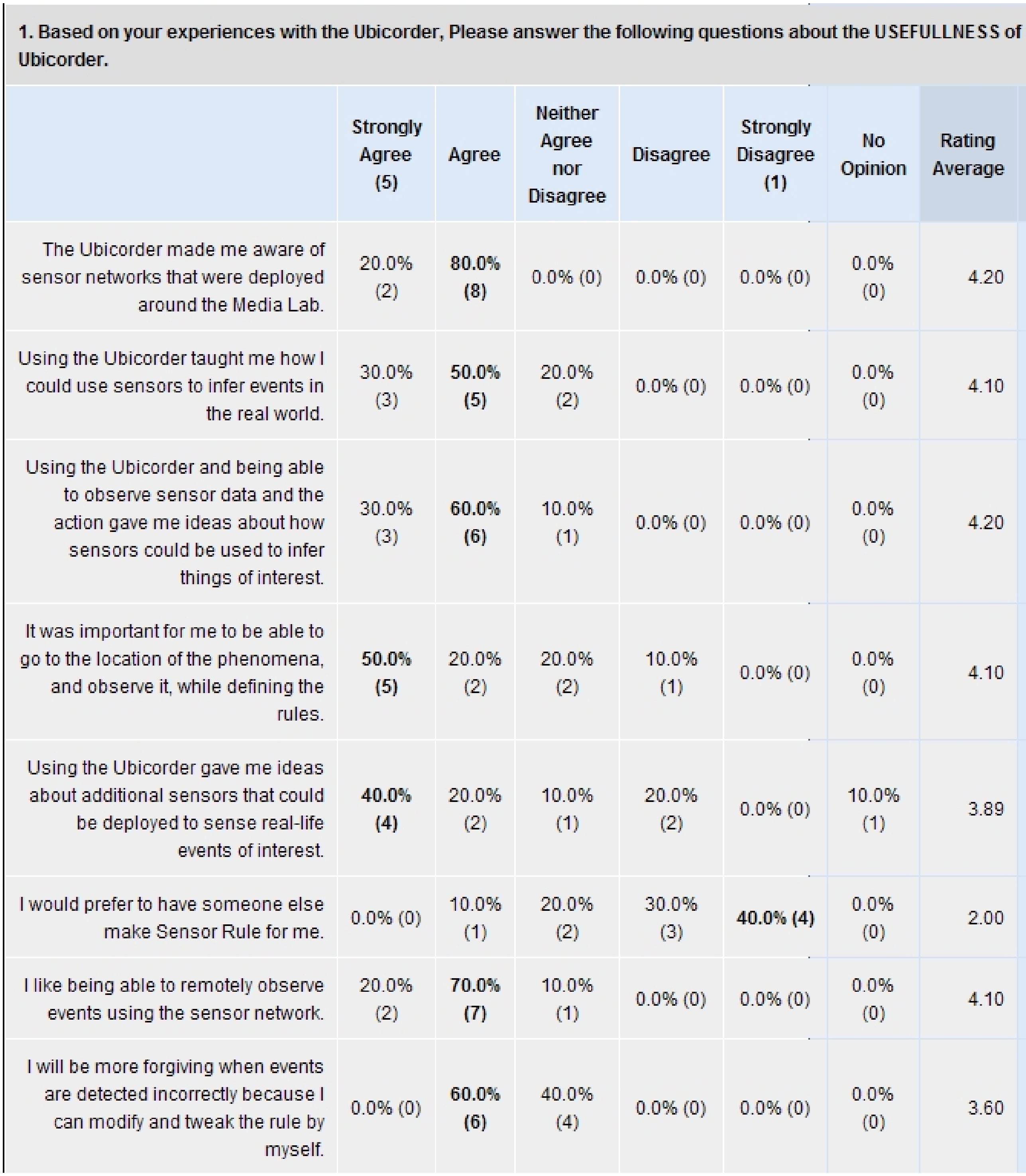
2. Simple Rule A2: True when light level = light level with LED spotlight on, but tube light off.

3. Compound Rule A: When either Simple Rule A1 *OR* Simple Rule A2 is true.

The participants then designed a simple rule (Simple Rule B) corresponding to the sound level, and tied the compound rule A and the simple rule B via an *AND* operator.

6.4.3 Post-Completion Questionnaire (Likert Scale Evaluation)

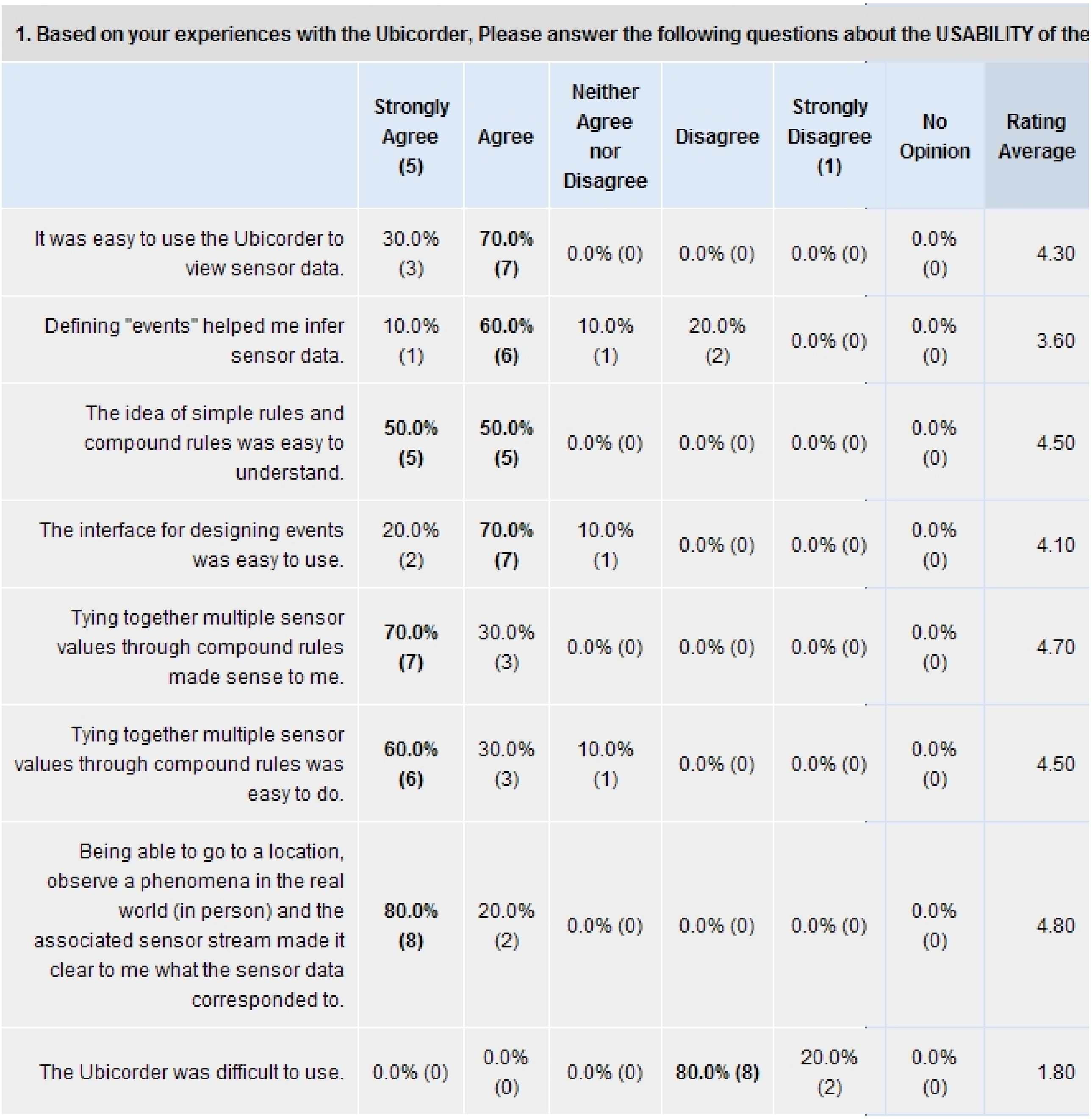
A post-completion questionnaire was given to the participants. Questions were asked about the usefulness of the Ubicorder (Table: 6-7), the device's usability (Table: 6-6) and the participants attitude toward the device and sensor networks (Table: 6-8).



1. Based on your experiences with the Ubicorder, Please answer the following questions about the USEFULLNESS of Ubicorder.

	Strongly Agree (5)	Agree	Neither Agree nor Disagree	Disagree	Strongly Disagree (1)	No Opinion	Rating Average
The Ubicorder made me aware of sensor networks that were deployed around the Media Lab.	20.0% (2)	**80.0% (8)**	0.0% (0)	0.0% (0)	0.0% (0)	0.0% (0)	4.20
Using the Ubicorder taught me how I could use sensors to infer events in the real world.	30.0% (3)	**50.0% (5)**	20.0% (2)	0.0% (0)	0.0% (0)	0.0% (0)	4.10
Using the Ubicorder and being able to observe sensor data and the action gave me ideas about how sensors could be used to infer things of interest.	30.0% (3)	**60.0% (6)**	10.0% (1)	0.0% (0)	0.0% (0)	0.0% (0)	4.20
It was important for me to be able to go to the location of the phenomena, and observe it, while defining the rules.	**50.0% (5)**	20.0% (2)	20.0% (2)	10.0% (1)	0.0% (0)	0.0% (0)	4.10
Using the Ubicorder gave me ideas about additional sensors that could be deployed to sense real-life events of interest.	**40.0% (4)**	20.0% (2)	10.0% (1)	20.0% (2)	0.0% (0)	10.0% (1)	3.89
I would prefer to have someone else make Sensor Rule for me.	0.0% (0)	10.0% (1)	20.0% (2)	30.0% (3)	**40.0% (4)**	0.0% (0)	2.00
I like being able to remotely observe events using the sensor network.	20.0% (2)	**70.0% (7)**	10.0% (1)	0.0% (0)	0.0% (0)	0.0% (0)	4.10
I will be more forgiving when events are detected incorrectly because I can modify and tweak the rule by myself.	0.0% (0)	**60.0% (6)**	40.0% (4)	0.0% (0)	0.0% (0)	0.0% (0)	3.60

Figure 6-6: Usefulness of the Ubicorder



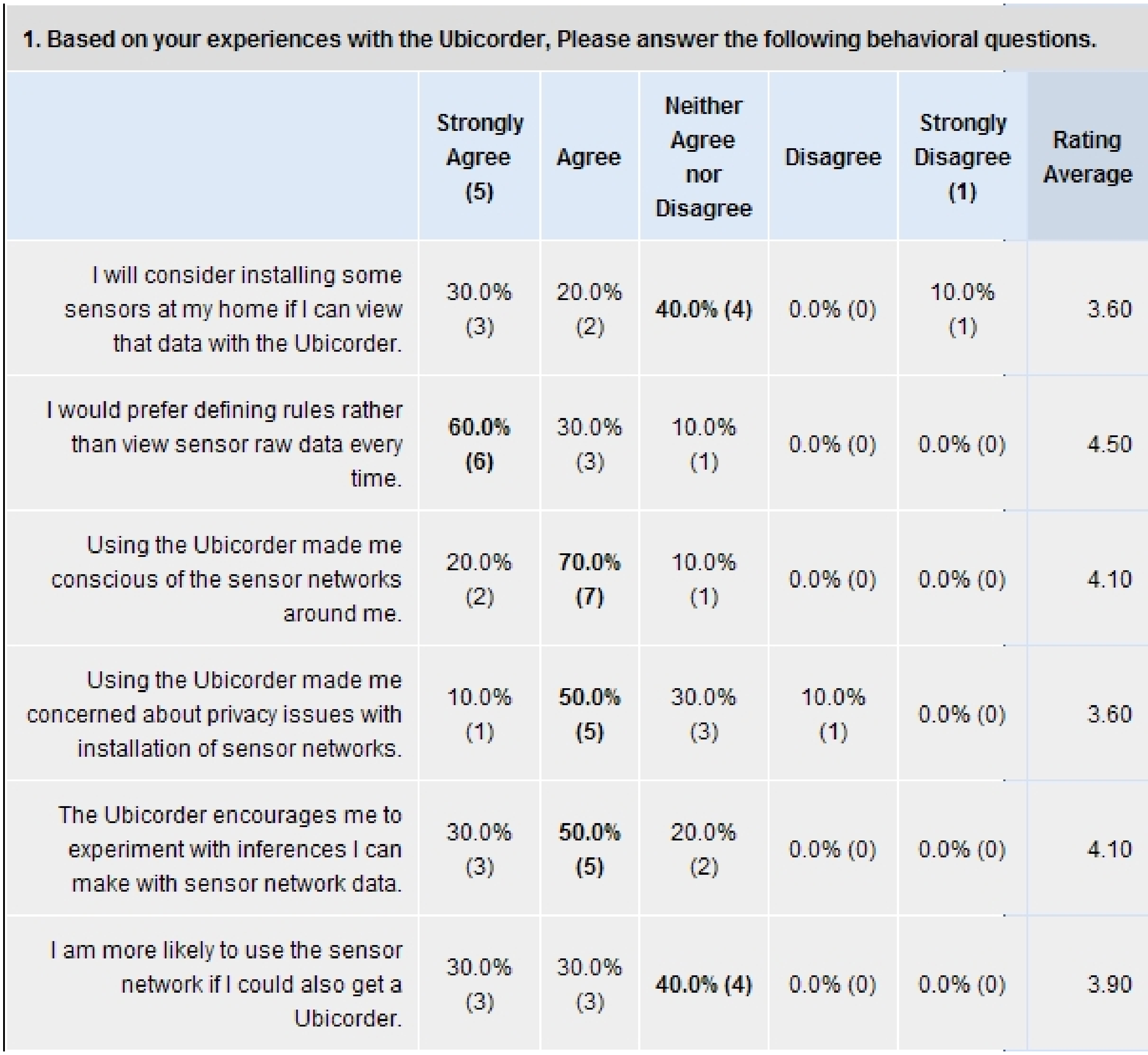
1. Based on your experiences with the Ubicorder, Please answer the following questions about the USABILITY of the

	Strongly Agree (5)	Agree	Neither Agree nor Disagree	Disagree	Strongly Disagree (1)	No Opinion	Rating Average
It was easy to use the Ubicorder to view sensor data.	30.0% (3)	**70.0% (7)**	0.0% (0)	0.0% (0)	0.0% (0)	0.0% (0)	4.30
Defining "events" helped me infer sensor data.	10.0% (1)	**60.0% (6)**	10.0% (1)	20.0% (2)	0.0% (0)	0.0% (0)	3.60
The idea of simple rules and compound rules was easy to understand.	**50.0% (5)**	**50.0% (5)**	0.0% (0)	0.0% (0)	0.0% (0)	0.0% (0)	4.50
The interface for designing events was easy to use.	20.0% (2)	**70.0% (7)**	10.0% (1)	0.0% (0)	0.0% (0)	0.0% (0)	4.10
Tying together multiple sensor values through compound rules made sense to me.	**70.0% (7)**	30.0% (3)	0.0% (0)	0.0% (0)	0.0% (0)	0.0% (0)	4.70
Tying together multiple sensor values through compound rules was easy to do.	**60.0% (6)**	30.0% (3)	10.0% (1)	0.0% (0)	0.0% (0)	0.0% (0)	4.50
Being able to go to a location, observe a phenomena in the real world (in person) and the associated sensor stream made it clear to me what the sensor data corresponded to.	**80.0% (8)**	20.0% (2)	0.0% (0)	0.0% (0)	0.0% (0)	0.0% (0)	4.80
The Ubicorder was difficult to use.	0.0% (0)	0.0% (0)	0.0% (0)	**80.0% (8)**	20.0% (2)	0.0% (0)	1.80

Figure 6-7: Usefulness of the Ubicorder

6.4.4 Post-Completion Questionnaire (Subjective Evaluation)

In this part of the questionnaire, participants were asked open-ended essay type questions about the Ubicorder. Their answers are in Figures: 6-9, 6-10, 6-11, 6-12, 6-13.



1. Based on your experiences with the Ubicorder, Please answer the following behavioral questions.

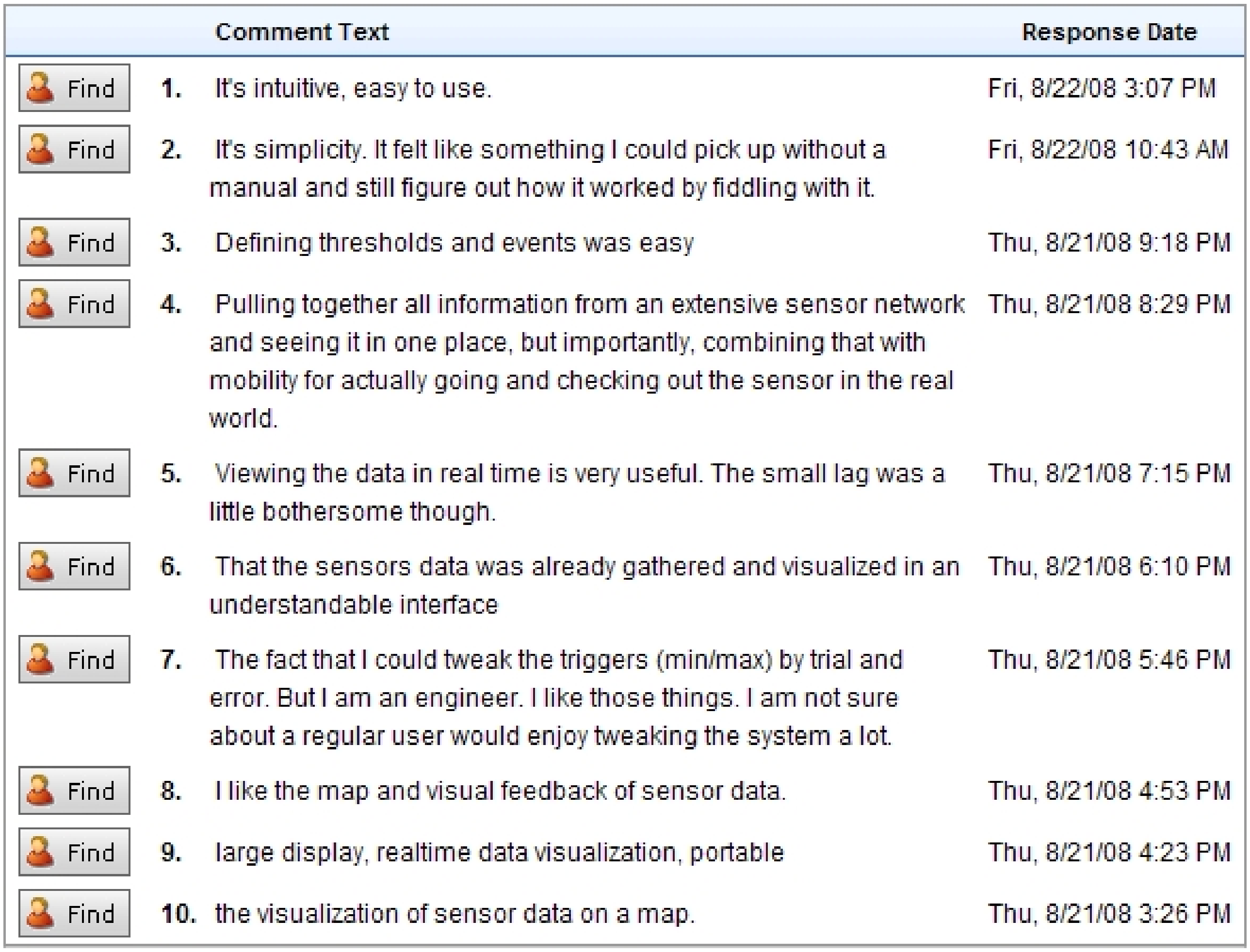
	Strongly Agree (5)	Agree	Neither Agree nor Disagree	Disagree	Strongly Disagree (1)	Rating Average
I will consider installing some sensors at my home if I can view that data with the Ubicorder.	30.0% (3)	20.0% (2)	**40.0% (4)**	0.0% (0)	10.0% (1)	3.60
I would prefer defining rules rather than view sensor raw data every time.	**60.0% (6)**	30.0% (3)	10.0% (1)	0.0% (0)	0.0% (0)	4.50
Using the Ubicorder made me conscious of the sensor networks around me.	20.0% (2)	**70.0% (7)**	10.0% (1)	0.0% (0)	0.0% (0)	4.10
Using the Ubicorder made me concerned about privacy issues with installation of sensor networks.	10.0% (1)	**50.0% (5)**	30.0% (3)	10.0% (1)	0.0% (0)	3.60
The Ubicorder encourages me to experiment with inferences I can make with sensor network data.	30.0% (3)	**50.0% (5)**	20.0% (2)	0.0% (0)	0.0% (0)	4.10
I am more likely to use the sensor network if I could also get a Ubicorder.	30.0% (3)	30.0% (3)	**40.0% (4)**	0.0% (0)	0.0% (0)	3.90

Figure 6-8: General questions about the Ubicorder, and sensor networks

6.5 Successes

6.5.1 Browsing

The Ubicorder successfully enabled participants with no prior sensor network experience to discover and explore the deployed sensor networks. Further, participants were able to browse and interpret the data gleaned by the networks.



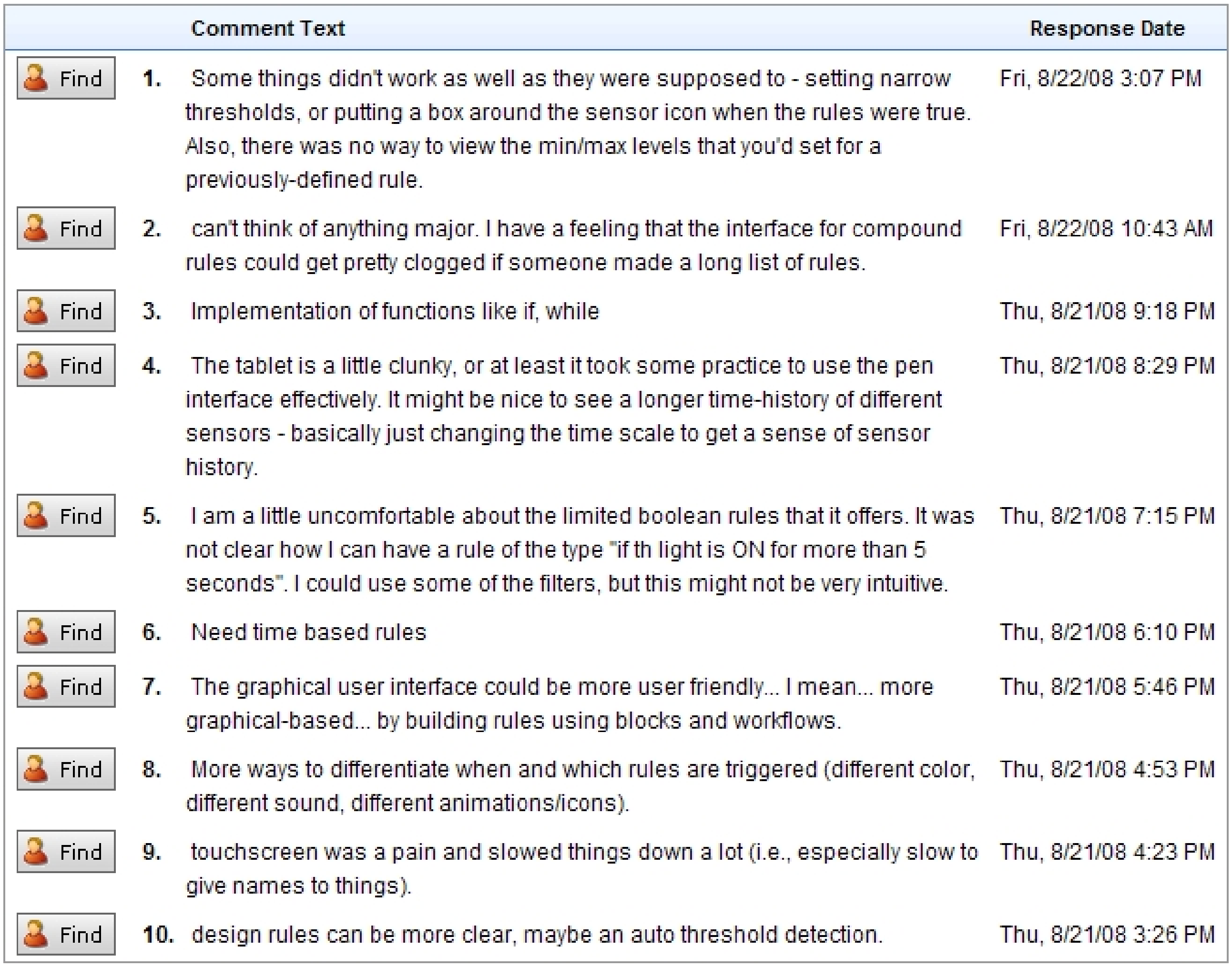
		Comment Text	Response Date
Find	1.	It's intuitive, easy to use.	Fri, 8/22/08 3:07 PM
Find	2.	It's simplicity. It felt like something I could pick up without a manual and still figure out how it worked by fiddling with it.	Fri, 8/22/08 10:43 AM
Find	3.	Defining thresholds and events was easy	Thu, 8/21/08 9:18 PM
Find	4.	Pulling together all information from an extensive sensor network and seeing it in one place, but importantly, combining that with mobility for actually going and checking out the sensor in the real world.	Thu, 8/21/08 8:29 PM
Find	5.	Viewing the data in real time is very useful. The small lag was a little bothersome though.	Thu, 8/21/08 7:15 PM
Find	6.	That the sensors data was already gathered and visualized in an understandable interface	Thu, 8/21/08 6:10 PM
Find	7.	The fact that I could tweak the triggers (min/max) by trial and error. But I am an engineer. I like those things. I am not sure about a regular user would enjoy tweaking the system a lot.	Thu, 8/21/08 5:46 PM
Find	8.	I like the map and visual feedback of sensor data.	Thu, 8/21/08 4:53 PM
Find	9.	large display, realtime data visualization, portable	Thu, 8/21/08 4:23 PM
Find	10.	the visualization of sensor data on a map.	Thu, 8/21/08 3:26 PM

Figure 6-9: Essay type answers to “What do you like most about the Ubicorder”

6.5.2 Defining, Manipulating and Using Rules

All participants were able to complete the tasks. The rules designed spanned a range of solutions. Further, once the participants were familiar with the basic interface of Ubicorder, the majority of their remaining time was spent exploring the sensor modalities, defining rules and experimenting with combinations of them. This rapid iteration of event rules enabled the participants to define robust rules.

Participants also liked the idea of using higher-level rules, and of giving them explicit control over the design of such rules. A majority of participants (70%) preferred that they (and not someone else) design the rules for themselves. Further, partic-



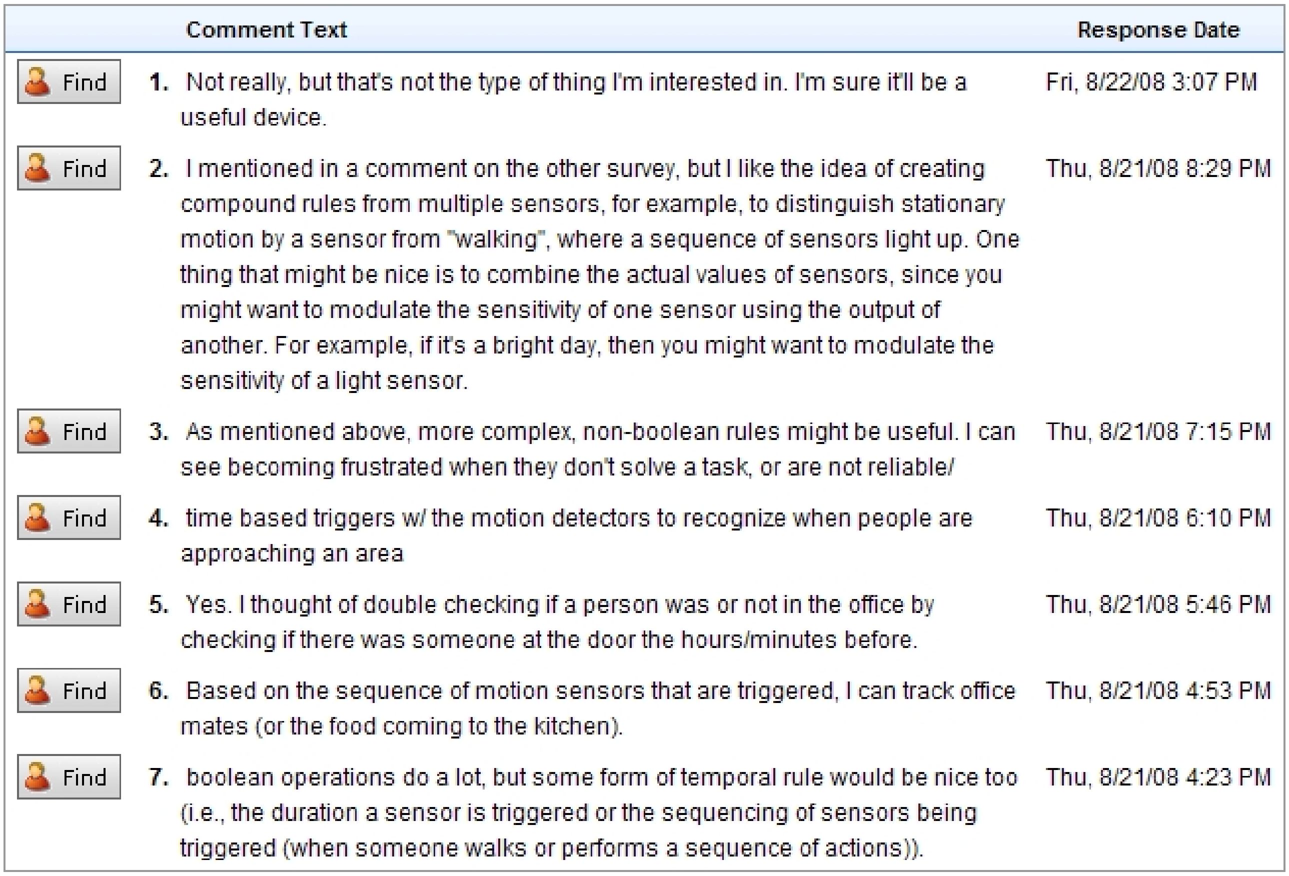
		Comment Text	Response Date
Find	1.	Some things didn't work as well as they were supposed to - setting narrow thresholds, or putting a box around the sensor icon when the rules were true. Also, there was no way to view the min/max levels that you'd set for a previously-defined rule.	Fri, 8/22/08 3:07 PM
Find	2.	can't think of anything major. I have a feeling that the interface for compound rules could get pretty clogged if someone made a long list of rules.	Fri, 8/22/08 10:43 AM
Find	3.	Implementation of functions like if, while	Thu, 8/21/08 9:18 PM
Find	4.	The tablet is a little clunky, or at least it took some practice to use the pen interface effectively. It might be nice to see a longer time-history of different sensors - basically just changing the time scale to get a sense of sensor history.	Thu, 8/21/08 8:29 PM
Find	5.	I am a little uncomfortable about the limited boolean rules that it offers. It was not clear how I can have a rule of the type "if th light is ON for more than 5 seconds". I could use some of the filters, but this might not be very intuitive.	Thu, 8/21/08 7:15 PM
Find	6.	Need time based rules	Thu, 8/21/08 6:10 PM
Find	7.	The graphical user interface could be more user friendly... I mean... more graphical-based... by building rules using blocks and workflows.	Thu, 8/21/08 5:46 PM
Find	8.	More ways to differentiate when and which rules are triggered (different color, different sound, different animations/icons).	Thu, 8/21/08 4:53 PM
Find	9.	touchscreen was a pain and slowed things down a lot (i.e., especially slow to give names to things).	Thu, 8/21/08 4:23 PM
Find	10.	design rules can be more clear, maybe an auto threshold detection.	Thu, 8/21/08 3:26 PM

Figure 6-10: Essay type answers to "What do you not like about the Ubicorder"

ipants, on average, were more forgiving when a rule based classification reported erroneous results, if they were given the ability to modify the rule.

6.5.3 Learning about Sensors and Sensor Networks

As reported in the questionnaire, the Ubicorder was useful in learning about both the sensor networks, and the sensors themselves. Participants liked being able to visually correlate actions in real life with sensor data observed on the Ubicorder. This activity can be classified as "learning by doing".



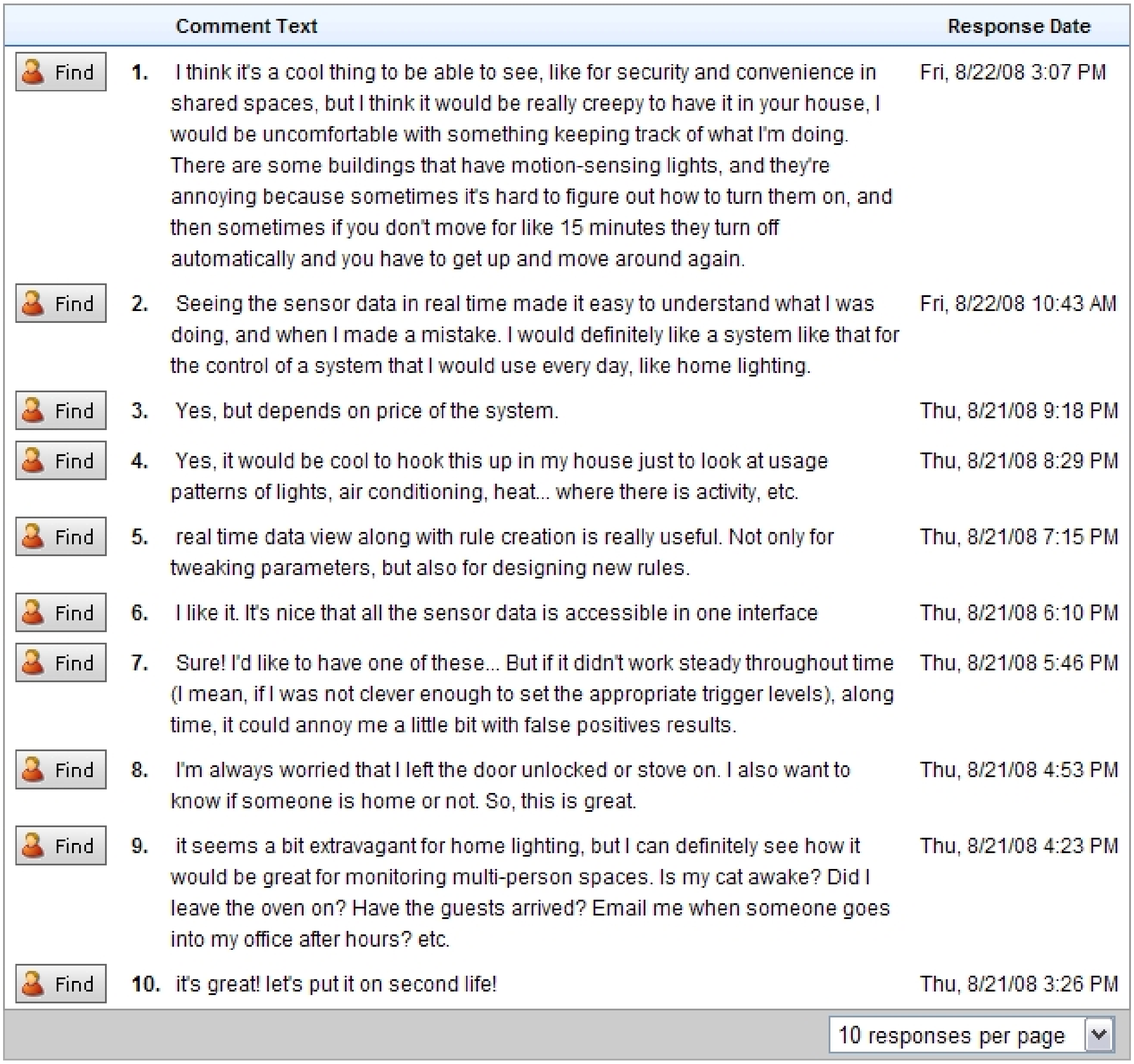
	Comment Text	Response Date
Find	1. Not really, but that's not the type of thing I'm interested in. I'm sure it'll be a useful device.	Fri, 8/22/08 3:07 PM
Find	2. I mentioned in a comment on the other survey, but I like the idea of creating compound rules from multiple sensors, for example, to distinguish stationary motion by a sensor from "walking", where a sequence of sensors light up. One thing that might be nice is to combine the actual values of sensors, since you might want to modulate the sensitivity of one sensor using the output of another. For example, if it's a bright day, then you might want to modulate the sensitivity of a light sensor.	Thu, 8/21/08 8:29 PM
Find	3. As mentioned above, more complex, non-boolean rules might be useful. I can see becoming frustrated when they don't solve a task, or are not reliable/	Thu, 8/21/08 7:15 PM
Find	4. time based triggers w/ the motion detectors to recognize when people are approaching an area	Thu, 8/21/08 6:10 PM
Find	5. Yes. I thought of double checking if a person was or not in the office by checking if there was someone at the door the hours/minutes before.	Thu, 8/21/08 5:46 PM
Find	6. Based on the sequence of motion sensors that are triggered, I can track office mates (or the food coming to the kitchen).	Thu, 8/21/08 4:53 PM
Find	7. boolean operations do a lot, but some form of temporal rule would be nice too (i.e., the duration a sensor is triggered or the sequencing of sensors being triggered (when someone walks or performs a sequence of actions)).	Thu, 8/21/08 4:23 PM

Figure 6-11: Essay type answers to "Did using the Ubicorder give you new ideas for inference rules that you could design ? If so, please mention some below"

6.5.4 Interface

Participants found the interface simple to use and intuitive. Furthermore participants liked that the Ubicorder was portable, and that they could design rules *in-situ*.

Participants also noted the importance of being able to be physically present at the location of the action/phenomena for which they were designing a rule. Mobility allowed participants to correlate the sensor signature with the real event, and was often cited as the feature they like the most.



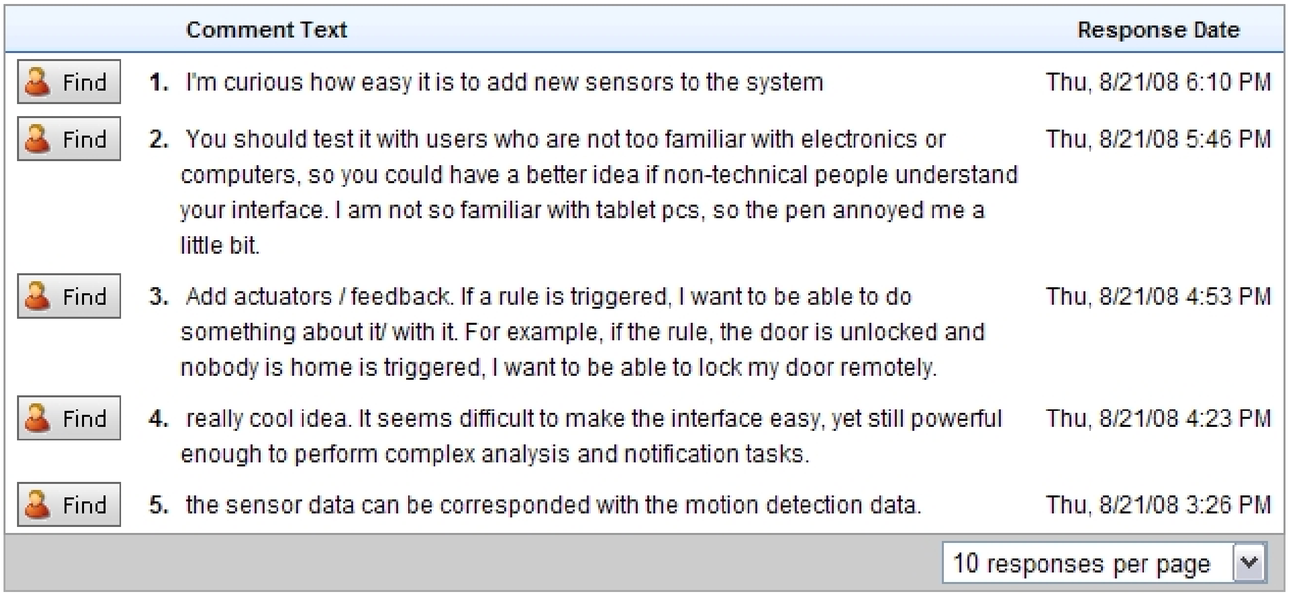
	Comment Text	Response Date
1.	I think it's a cool thing to be able to see, like for security and convenience in shared spaces, but I think it would be really creepy to have it in your house, I would be uncomfortable with something keeping track of what I'm doing. There are some buildings that have motion-sensing lights, and they're annoying because sometimes it's hard to figure out how to turn them on, and then sometimes if you don't move for like 15 minutes they turn off automatically and you have to get up and move around again.	Fri, 8/22/08 3:07 PM
2.	Seeing the sensor data in real time made it easy to understand what I was doing, and when I made a mistake. I would definitely like a system like that for the control of a system that I would use every day, like home lighting.	Fri, 8/22/08 10:43 AM
3.	Yes, but depends on price of the system.	Thu, 8/21/08 9:18 PM
4.	Yes, it would be cool to hook this up in my house just to look at usage patterns of lights, air conditioning, heat... where there is activity, etc.	Thu, 8/21/08 8:29 PM
5.	real time data view along with rule creation is really useful. Not only for tweaking parameters, but also for designing new rules.	Thu, 8/21/08 7:15 PM
6.	I like it. It's nice that all the sensor data is accessible in one interface	Thu, 8/21/08 6:10 PM
7.	Sure! I'd like to have one of these... But if it didn't work steady throughout time (I mean, if I was not clever enough to set the appropriate trigger levels), along time, it could annoy me a little bit with false positives results.	Thu, 8/21/08 5:46 PM
8.	I'm always worried that I left the door unlocked or stove on. I also want to know if someone is home or not. So, this is great.	Thu, 8/21/08 4:53 PM
9.	it seems a bit extravagant for home lighting, but I can definitely see how it would be great for monitoring multi-person spaces. Is my cat awake? Did I leave the oven on? Have the guests arrived? Email me when someone goes into my office after hours? etc.	Thu, 8/21/08 4:23 PM
10.	it's great! let's put it on second life!	Thu, 8/21/08 3:26 PM

Figure 6-12: Essay type answers to “What do you think about the idea of being able to see, in real-time, the sensor data collected. Would you want such a system, to say, control your home lighting?”

6.6 Shortcomings

6.6.1 Limitations and Critical Analysis of the User Study

There were some limitations of the user study in the present form:



	Comment Text	Response Date
Find	1. I'm curious how easy it is to add new sensors to the system	Thu, 8/21/08 6:10 PM
Find	2. You should test it with users who are not too familiar with electronics or computers, so you could have a better idea if non-technical people understand your interface. I am not so familiar with tablet pcs, so the pen annoyed me a little bit.	Thu, 8/21/08 5:46 PM
Find	3. Add actuators / feedback. If a rule is triggered, I want to be able to do something about it/ with it. For example, if the rule, the door is unlocked and nobody is home is triggered, I want to be able to lock my door remotely.	Thu, 8/21/08 4:53 PM
Find	4. really cool idea. It seems difficult to make the interface easy, yet still powerful enough to perform complex analysis and notification tasks.	Thu, 8/21/08 4:23 PM
Find	5. the sensor data can be corresponded with the motion detection data.	Thu, 8/21/08 3:26 PM

10 responses per page

Figure 6-13: Essay type answers to "Any additional comments, and last thoughts".

- Time dependent combination rules were not tested: Several users felt the need for defining compound rules that temporally linked other rules. The Ubicorder does indeed incorporate time based rules, but such functionality was not explained or expected of the user in the given user study.

- Nature of the Task: The task was very clearly defined for the user, i.e., make a rule for detecting when the microscope/soldering station is in use. Ideally, the participant should independently discover new application spaces based on the sensors available. However, such an open ended task is not well suited for a short-term user study.

- Browsing interface: The Ubicorder can be better tested for its functionality as a sensor network browsing device. In particular, it would be interesting to use the browser on a large scale, multi-sensor deployment.

6.6.2 Shortcomings of the System

Participants identified some key areas for improvement.

- Use of “hard” boolean operators to tie together component rules (to form compound ones). Limiting rules to boolean operators seemed to be restrict the expressivity of the system.

- The system currently requires the user to create the rule, the user creates and tweaks the rule. Participants suggested that a “first guess” from the system will be useful.

- Participants pointed out the need for categorization of the (previously made) component rules in the list of rules displayed when the user designs a compound rule. Currently, all previously designed rules are displayed as a vertical list. The rules appear in the order of their creation, i.e., the oldest rule is on the top, the newest rule is at the bottom. Participants suggested the need to aid navigation of these existing rules based on participating nodes or sensors.

- Participants pointed to the need for more expressive, customizable alerts / actuation (i.e., steps to be taken when the rule is true). One participant suggested that users be allowed to put a specific image/photograph on the floor plan, when a rule is true. For example, if a rule corresponding to Joe’s presence in office is true, the system should be able to display Joe’s photograph on the floor plan (that is displayed in the browse mode). Another participant mentioned that they would like the events to actuate something physical, for example, unlock a door. This functionality can currently be achieved by sending operating system keystrokes (which the Ubicorder allows) and writing a stub to talk to external devices.

- Participants found using the touchscreen cumbersome, especially when entering the rule name. In the future, we propose that the system “suggest” to the users a name, perhaps based on the sensing modality and the sensor node.

- One participant (Figure: 6-9, Q. 5) commented upon the latency of the system, finding it a "little bothersome". In the present setup, the majority of the latency was because of the smoothing filter used while rule creation. The smoothing filter works by taking a moving average. There is no mechanism to eliminate this latency. One alternative would be to display to the user both the unfiltered, real-time data stream and the filtered output, simultaneously.

- One participant (an adept programmer) commented on the (perceived) lack of implementation of functions like if, while. The EDITY system does indeed incorporate these mechanisms; it present them as edge-triggered (if) and level-triggered actions (while). It illustrates the challenge when users translate terms from programming domain to signal processing space.

Several of these shortcomings are further discussed in Section: 7.1.

Chapter 7

Conclusions and Future Work

Through the work presented in this thesis, a new research field has emerged. This field sits at the crossroads of Human Computer Interaction, Sensor Networks and Ubiquitous Computing.

We have illustrated that it is possible to devise systems that quickly and easily makes sensor networks accessible to end users. Our results indicate that such a system promotes the exploration and use of available sensor networks.

Our results indicate that developing better interfaces to support end user inference and understanding of the sensor network encourages an increased use of sensor networks. Users preferred defining their own rules, in contrast to relying on predefined rules.

By increasing exposure to the sensor networks, the Ubicorder encourages users to experiment with the facilities of the networks and to find new, compelling application scenarios.

7.1 Future Work

The Ubicorder may be improved in several ways. These improvements were identified based on feedback from the users tests, and insights gleaned by having designed the system. Some of these have also been discussed briefly in Section: 6.6.2.

7.1.1 Smaller, Lighter, and Faster

The Ubicorder's current implementation uses a 12.1" tablet as the base computer. Moving the Ubicorder to a smaller form factor computer, for example a PDA sized device would make it more convenient for the user. The primary challenge of porting to a small form-factor device is the limited screen real-estate. While the Ubicorder's browsing functionality can be easily ported, the EDITY interface will require more changes. One idea is to make each pane of the EDITY UI as a separate screen, and the user can scroll through the three panes one at a time.

Some users currently complained of the latency between a physical phenomena and its observation on the Ubicorder. While a part of the latency is due to the network and application of signal processing filters (which delay change by smoothing them out), displaying and plotting the results also takes considerable computation. Moving from a Java based implementation to a C++ based implementation will speed up the graphing and visualization process.

7.1.2 Suggesting Rules

The Ubicorder would be easier to use if it suggested an "initial guess" or the rule. The user can then tweak such a guess. In contrast, the current interface relies entirely upon the end user for the creation of rules. One way to incorporate such a first guess would be to display the maximum/minimum threshold of the sensor data currently being displayed.

Another idea is to have a system that automatically detects the changes in state. The user merely marks "Now" and "Not Now" regions in the time series graph. The system computes the features that will serve as the best discriminants to differentiate this rule from all other existing rules. As an example, consider the follow use-case.

- User is trying to create a rule. He uses the rule creation interface and selects - "create by example".
- User Switches on the light. User clicks the button "Record State On".
- User Switches off the light. User clicks the button "Record State Off"
- The system tries to determine by this example what could serve as discriminant between the state. The system determines that the light level in a particular sensor varied, and uses this as a rule.
- The system then displays a list of sensors that varied, and the rules it can derive from them. This information may be displayed to the user, and the user may be allowed to manipulate it.

Another important addition can be a mechanism to suggest to the user how some rules can be combined. Again, the approach to "record sample" could be used, with multiple samples leading to either relaxing/modifying the thresholds for a rule, or creating a compound rule by internally "OR"ing the scenarios. Having a system help users generalize their rule would be an extremely useful feature.

Additional ideas for aiding the user to design compound rules are discussed below (Section: 7.1.5).

7.1.3 Expressivity of Rules

The use of "hard" boolean operators to tie together component rules (to form compound ones) limits expressivity of the system. Future versions of the Ubicorder will incorporate a probabilistic model for combining rules. A fuzzy logic based approach may be suitable for the given application.

More expressive "output" or event metaphors can be incorporated in the next version of the Ubicorder. We currently support a limited set of actuators, i.e., screen actions to indicate that the rule is true. However, given that the designer can not envision all possible application scenarios the system might be used for, we propose a plug-in based extensible work. For example, the “image” plug-in might support the display of a person’s photograph every time a rule testing for the occupancy of an office is detected as true.

7.1.4 Generalizing the Interface

The model presented in this thesis is general, although the current instantiation (this implementation) is not. Additional filters and operators can extend the functionality of the system. For example, operators such as “Highly Correlated with” instead of the boolean operators would be needed.

7.1.5 Interface

We had hoped that the user would experiment with boolean operators for the design of compound rules, finding the right one after a few tries. Instead, we found that users did not experiment with the boolean operators, but instead choose one based on their mental model of the system. Such an approach could lead to the user missing a simpler set of rules. In the next version of the Ubicorder, we intend to display the output of all the boolean operators simultaneously, i.e., we display the AND, the OR and the XOR outputs in parallel, on three different result strip charts. The user would then select the correct result strip chart based on the desired output. This would encourage matching the right boolean operator by experimentation.

When designing a compound rule, choosing the right component rules for combination proved to be a difficult task. It would be useful to see the output of all possible candidate rules simultaneously, and then choose the right ones. Currently, the only way to see the output of a (candidate) rule is to select it. However, selecting the rule means that it forms a component rule of the compound rule being designed. Instead, we hope to convey the output of the displayed rule by color coding the rule in the left pane (list of candidate rules). For example, the button corresponding to the rule can be of the color red or green, indicating the rule's present output. Such an interface will enable the user to see the state of the entire set of previously created rules at a glance.

Another area of improvement is the descriptiveness of the rules. Currently, the rule created is only described by its name. However, it would be useful to record a description of why the user created or modified the rule, while they are modifying it. This information could be especially useful when another user is assembling a compound rule that uses an older rule. Perhaps a text box should be incorporated in the interface to store the comments associated with the rule, and this information is displayed when the user tweaks the rule/browses it as a candidate for a component rule.

Currently, the Ubicorder's EDITY system does not allow for selection of multiple sensor nodes, and then operate on all the nodes together. Some sample operations would be:

- Any of the selected sensors: Rules such as "true if any of the sensors in the building report temperature $\geq$ 80 degrees".
- Aggregate of selected sensor nodes: Rules such as "true if the aggregate of temperature readings from all sensors $\geq$ 80 degrees".

Such rules can be easily incorporated in the Ubicorder's interface.

Lastly, Participants found using the touchscreen cumbersome, especially when entering the rule name. In the future, we propose that the system "suggest" to the users a name, perhaps based on the location of the user.

7.1.6 Building a Community

Future versions of the Ubicorder can facilitate and encourage sharing of rules across multiple users and devices. One can consider instituting a Wikipedia [1] like repository of user created sensor rules. These rules may be generic, or location-specific. For example, there could be multiple rules in the repository intended detect if a person is in front of a white board if a motion sensor is mounted at an 45 degree angle at the top. A new user with a similar (not necessarily identical) sensor arrangement could download one of the many rules and use the one which works best. Additionally, the rules could be more location specific, for example, the rule to detect if the there is a meeting going on in a conference room B in building E15. Many users, with their own Ubicorder, might have attempted to create the rule, but the best one will "bubble" up. The repository can also let users rank the rules for their effectiveness, allowing the best rules to emerge.

The user could also leave "breadcrumbs," i.e., inference rules that are displayed to the user when they are at a particular location. For example, a Ubicorder will automatically display the seat availability in a sensor equipped bus as it approaches a bus stop, if the Ubicorder user is waiting at the bus stop.

7.1.7 New Applications

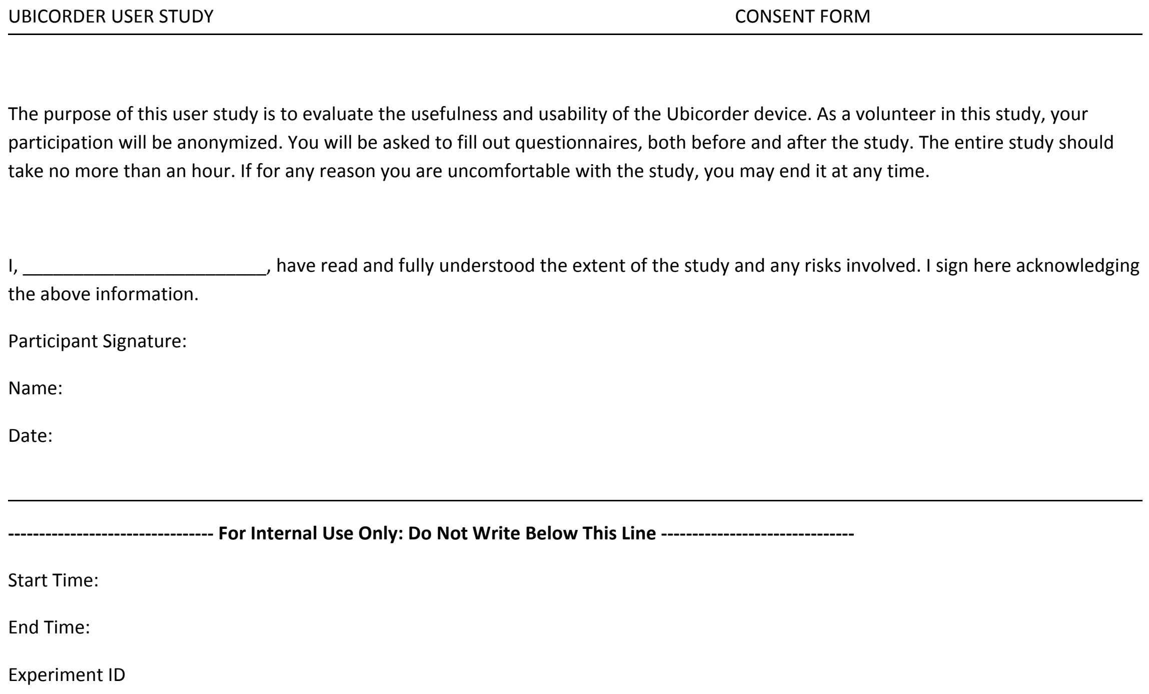
The Ubicorder can also serve as a powerful tool for exploring human attitudes and preferences towards sensor networks, and for addressing privacy concerns that straddle sensor networks deployed in home and office environments. For example, with the Ubicorder, a user could limit the granularity and the latency of sensor data exposed by a sensor node. Such limitations will translate to either coarser inferences and/or delayed inferences.

7.2 Concluding Remarks

If the vision of Ubiquitous Computing is to be realized, it is essential to enable end-users to use sensors and sensor networks. The Ubicorder aims to lower the threshold for users to discover, use and take advantage of the facilities offered by sensor networks, and thus, we believe, represents an important step in this direction.

Appendix A

User Study Material



UBICORDER USER STUDY CONSENT FORM

The purpose of this user study is to evaluate the usefulness and usability of the Ubicorder device. As a volunteer in this study, your participation will be anonymized. You will be asked to fill out questionnaires, both before and after the study. The entire study should take no more than an hour. If for any reason you are uncomfortable with the study, you may end it at any time.

I, _________________________, have read and fully understood the extent of the study and any risks involved. I sign here acknowledging the above information.

Participant Signature:

Name:

Date:

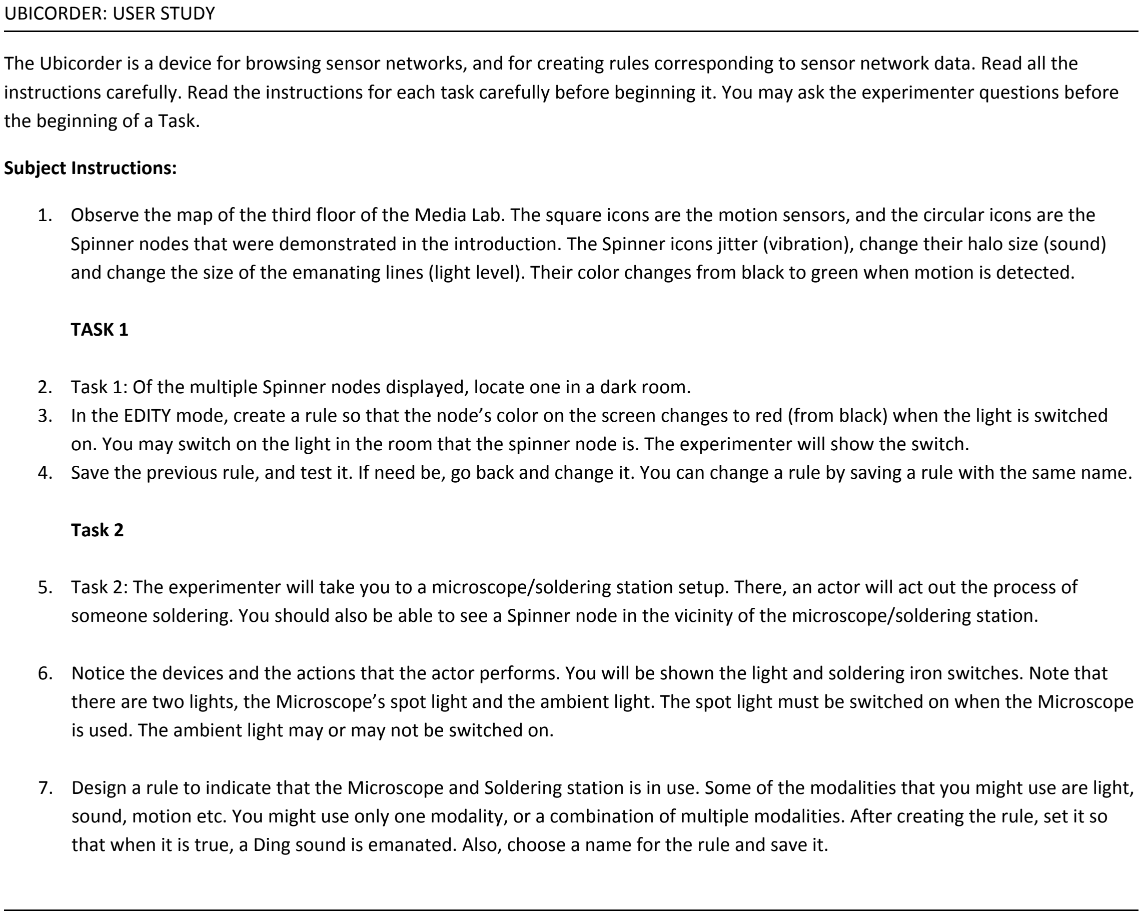
--------------------------------- **For Internal Use Only: Do Not Write Below This Line** -------------------------------

Start Time:

End Time:

Experiment ID

Figure A-1: Consent Form



UBICORDER: USER STUDY

The Ubicorder is a device for browsing sensor networks, and for creating rules corresponding to sensor network data. Read all the instructions carefully. Read the instructions for each task carefully before beginning it. You may ask the experimenter questions before the beginning of a Task.

Subject Instructions:

1. Observe the map of the third floor of the Media Lab. The square icons are the motion sensors, and the circular icons are the Spinner nodes that were demonstrated in the introduction. The Spinner icons jitter (vibration), change their halo size (sound) and change the size of the emanating lines (light level). Their color changes from black to green when motion is detected.

   TASK 1

2. Task 1: Of the multiple Spinner nodes displayed, locate one in a dark room.
3. In the EDITY mode, create a rule so that the node's color on the screen changes to red (from black) when the light is switched on. You may switch on the light in the room that the spinner node is. The experimenter will show the switch.
4. Save the previous rule, and test it. If need be, go back and change it. You can change a rule by saving a rule with the same name.

   Task 2

5. Task 2: The experimenter will take you to a microscope/soldering station setup. There, an actor will act out the process of someone soldering. You should also be able to see a Spinner node in the vicinity of the microscope/soldering station.

6. Notice the devices and the actions that the actor performs. You will be shown the light and soldering iron switches. Note that there are two lights, the Microscope's spot light and the ambient light. The spot light must be switched on when the Microscope is used. The ambient light may or may not be switched on.

7. Design a rule to indicate that the Microscope and Soldering station is in use. Some of the modalities that you might use are light, sound, motion etc. You might use only one modality, or a combination of multiple modalities. After creating the rule, set it so that when it is true, a Ding sound is emanated. Also, choose a name for the rule and save it.

Figure A-2: Instructions to participants

Appendix B

Firmware: USB Code (C)

```c
/* This header file is part of the ATMEL AVR32-SoftwareFramework-1.3.0-AT32UC3B Release */

/*This file is prepared for Doxygen automatic documentation generation.*/
/*! \file *********************************************************************
 *
 * \brief Management of the USB device mouse CDC task.
 *
 * This file manages the USB device mouse CDC task.
 *
 * - Compiler:           IAR EWAVR32 and GNU GCC for AVR32
 * - Supported devices:  All AVR32 devices with a USB module can be used.
 * - AppNote:
 *
 * \author               Atmel Corporation: http://www.atmel.com \n
 *                       Support and FAQ: http://support.atmel.no/
 *
 ******************************************************************************/

/* Copyright (C) 2006-2008, Atmel Corporation All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice,
 * this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright notice,
 * this list of conditions and the following disclaimer in the documentation
 * and/or other materials provided with the distribution.
 *
 * 3. The name of ATMEL may not be used to endorse or promote products derived
 * from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY ATMEL ``AS IS'' AND ANY EXPRESS OR IMPLIED
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY AND
 * SPECIFICALLY DISCLAIMED. IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT,
 * INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
 * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
 * ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
 * THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */


#ifndef _DEVICE_CDC_TASK_H_
#define _DEVICE_CDC_TASK_H_


//_____ I N C L U D E S ___________________________________________________

#include "conf_usb.h"

#if USB_DEVICE_FEATURE == DISABLED
  #error device_cdc_task.h is #included although USB_DEVICE_FEATURE is disabled
#endif


//_____ D E F I N I T I O N S _____________________________________________

#define NB_MS_BEFORE_FLUSH   50
#define POLL_COMPASS    'c'

//_____ M A C R O S _______________________________________________________


//_____ D E C L A R A T I O N S ___________________________________________

extern void device_cdc_task_init(void);
#ifdef FREERTOS_USED
extern void device_cdc_task(void *pvParameters);
#else
extern void device_cdc_task(void);
#endif
extern void usb_sof_action(void);

#endif  // _DEVICE_CDC_TASK_H_
```

```
/* This source file is part of the ATMEL AVR32-SoftwareFramework-1.3.0-AT32UC3B Release */

/*This file is prepared for Doxygen automatic documentation generation.*/
/*! \file *********************************************************************
 *
 * \brief Management of the USB device CDC task.
 *
 * This file manages the USB device CDC task.
 *
 * - Compiler:           IAR EWAVR32 and GNU GCC for AVR32
 * - Supported devices:  All AVR32 devices with a USB module can be used.
 * - AppNote:
 *
 * \author               Atmel Corporation: http://www.atmel.com \n
 *                       Support and FAQ: http://support.atmel.no/
 *
 ******************************************************************************/

/* Copyright (C) 2006-2008, Atmel Corporation All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice,
 * this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright notice,
 * this list of conditions and the following disclaimer in the documentation
 * and/or other materials provided with the distribution.
 *
 * 3. The name of ATMEL may not be used to endorse or promote products derived
 * from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY ATMEL ``AS IS'' AND ANY EXPRESS OR IMPLIED
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY AND
 * SPECIFICALLY DISCLAIMED. IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT,
 * INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
 * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
 * ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
 * THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */


//_____  I N C L U D E S ___________________________________________________

#include <stdio.h>
#include "usart.h"     // Shall be included before FreeRTOS header files, since 'inline' is defined to ''; leading to
                       // link errors

#include "conf_usb.h"
#define BUFSIZE 200

#if USB_DEVICE_FEATURE == ENABLED

#include "board.h"
#ifdef FREERTOS_USED
#include "FreeRTOS.h"
#include "task.h"
#endif
#include "usb_drv.h"
#include "gpio.h"
// #include "joystick.h"
#include "usb_descriptors.h"
#include "usb_standard_request.h"
#include "device_cdc_task.h"
#include "uart_usb_lib.h"
#include "shirt_compass.h"


//_____ M A C R O S ________________________________________________________


//_____ D E F I N I T I O N S ______________________________________________

//_____ D E C L A R A T I O N S ____________________________________________

static volatile U16  sof_cnt;



//!
//! @brief This function initializes the hardware/software resources
//! required for device CDC task.
//!
void device_cdc_task_init(void)
{
  sof_cnt   =0 ;
  uart_usb_init();

#ifndef FREERTOS_USED
  #if USB_HOST_FEATURE == ENABLED
  // If both device and host features are enabled, check if device mode is engaged
  // (accessing the USB registers of a non-engaged mode, even with load operations,
  // may corrupt USB FIFO data).
  if (Is_usb_device())
  #endif  // USB_HOST_FEATURE == ENABLED
    Usb_enable_sof_interrupt();
#endif  // FREERTOS_USED

#ifdef FREERTOS_USED
  xTaskCreate(device_cdc_task,
              configTSK_USB_DCDC_NAME,
              configTSK_USB_DCDC_STACK_SIZE,
              NULL,
              configTSK_USB_DCDC_PRIORITY,
              NULL);

#endif  // FREERTOS_USED
}
short ir_sensor_packet[17];

//!
//! @brief Entry point of the device CDC task management
//!
#ifdef FREERTOS_USED
void device_cdc_task(void *pvParameters)
#else
void device_cdc_task(void)
#endif
```

```
{

  int c;
  int i;
  char buffer[BUFSIZE];
  static Bool startup=TRUE;

#ifdef FREERTOS_USED
  portTickType xLastWakeTime;
  xLastWakeTime = xTaskGetTickCount();

  while (TRUE)
  {
    vTaskDelayUntil(&xLastWakeTime, configTSK_USB_DCDC_PERIOD);

    // First, check the device enumeration state
    if (!Is_device_enumerated()) continue;
#else
    // First, check the device enumeration state
    if (!Is_device_enumerated())
     return;
#endif  // FREERTOS_USED

    if( startup )
    {
       // printf("\r\nUSB DEVICE Communications Device Class demo.\r\n");
       startup=FALSE;
    }

    if( sof_cnt>=NB_MS_BEFORE_FLUSH )  //Flush buffer in Timeout
    {
       sof_cnt=0;
       uart_usb_flush();
    }

//    if ( is_joystick_right() )
//       printf("Joystick Right key pressed.\r\n");
//
//    if ( is_joystick_left() )
//       printf("Joystick Left key pressed.\r\n");
//
//    if ( is_joystick_down() )
//       printf("Joystick Down key pressed.\r\n");
//
//    if ( is_joystick_up() )
//       printf("Joystick Up key pressed.\r\n");
//
//    if ( is_joystick_pressed() )
//       printf("Joystick Select key pressed.\r\n");

//    if (!gpio_get_pin_value(GPIO_PUSH_BUTTON_0))
//       printf("Button 0 key pressed.\r\n");
//
//    if (!gpio_get_pin_value(GPIO_PUSH_BUTTON_1))
//       printf("Button 1 key pressed.\r\n");

//    if (usart_test_hit(DBG_USART))    // Something on USART ?
//    {
//       if( uart_usb_tx_ready() )      // "USART"-USB free ?
//       {
//          if( USART_SUCCESS==usart_read_char(DBG_USART, &c) )
//          {
//             if( c=='\r' )
//                uart_usb_putchar('\n');
//
//             uart_usb_putchar(c);           // Loop back, USART to USB
//             // LED_Toggle(LED0);
//          }
//          else {
//             usart_reset_status( DBG_USART );
//          }
//       }
//    }

    if (uart_usb_test_hit())          // Simple Loopback code - Echo's back whatever was sent over USB
    {
       if (uart_usb_tx_ready()) // USART free ?
       {
          c= uart_usb_getchar();

          switch(c) {

          default :
              if( c=='\r' )
                uart_usb_putchar('\n');
              uart_usb_putchar(c);   // Loop Back to USB
              break;

          case POLL_COMPASS :
           get_compass_data(POST_HEADING, buffer);
           uart_usb_putchar('\r');
           uart_usb_putchar('\n');
           for(i = 0; i < 6; i++) {
            uart_usb_putchar(buffer[i]);
           }
           uart_usb_putchar('\r');
           uart_usb_putchar('\n');
           break;

          case 'i':
           uart_usb_putchar('\r');
           uart_usb_putchar('\n');

//            short* packet = get_ir_packet();
           for(i = 0; i < 17; i++) {
            uart_usb_putchar(ir_sensor_packet[i]);
           }
           uart_usb_putchar('\r');
           uart_usb_putchar('\n');

           break;
          }
       }
    }
//    if (uart_usb_test_hit())          // Something received from the USB ?
//    {
//       if (usart_tx_ready(DBG_USART)) // USART free ?
//       {
//          c= uart_usb_getchar();
//          if( c=='\r' )
//            usart_putchar(DBG_USART, '\n');
//
//           usart_putchar(DBG_USART, c);   // loop back USB to USART
//           // LED_Toggle(LED1);
//       }
//    }


#ifdef FREERTOS_USED
  }
#endif
}


//!
//! @brief usb_sof_action
//!
//! This function increments the sof_cnt counter each time
//! the USB Start-of-Frame interrupt subroutine is executed (1 ms).
//! Useful to manage time delays
//!
void usb_sof_action(void)
{
  sof_cnt++;
}

#endif  // USB_DEVICE_FEATURE == ENABLED
```

Appendix C

EDITY Logic Code (Java)

```
/**                                                                                  }
 *                                                                                  }
 */
package edu.media.resenv.clairvoyant;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;

import org.eclipse.swt.widgets.Display;

/**
 *
 * Main class
 * Sets up state variables.
 *
 * Clairvoyant
 * Copyright (c) 2008 - 2010, MIT Media Laboratory
 * All Rights Reserved
 *
 *
 * @author Manas Mittal, MIT Media Laboratory
 *
 */

final public class Clairvoyant {
 int i;
 static volatile LinkedHashMap<String, Node> nodes; // does this need to be volatile
 // static Display display;

 public static void main(String[] args) {
  Clairvoyant.nodes = new LinkedHashMap<String, Node>();
  // Build a list of nodes
  new MerlSensorListParser((LinkedHashMap<String, Node>) nodes);

  // Open and start the compass (if connected)
  (new Thread (new ComPortFuncs())).start(); // change for N810

  //// start visualization
  try {
   (new Thread(new BigMapView(nodes))).start();
  }
  catch (Exception e) {
   e.printStackTrace();
  }

  // Start Threads for each Merl Node Sensor Parent
  (new Thread(new MerlListener(nodes, "beetle.media.mit.edu", 8484))).start();
  (new Thread(new MerlListener(nodes, "ant.media.mit.edu", 8484))).start();
  (new Thread(new MerlListener(nodes, "gnat.media.mit.edu", 8484))).start();
  (new Thread(new MerlListener(nodes, "slug.media.mit.edu", 8484))).start();
  (new Thread(new MerlListener(nodes, "firefly.media.mit.edu", 8484))).start();
  (new Thread(new MerlListener(nodes, "mantis.media.mit.edu", 8484))).start();
//  (new Thread(new SpinwerkListener(nodes, ”18.85.45.202”, 2001, new Point(120,90)))).start(); // DUMMY NODE
  (new Thread(new SpinwerkListener(nodes, "18.85.45.161", 1000, new Point(125,5)))).start(); // RESENV OFFICE NODE
  (new Thread(new SpinwerkListener(nodes, "18.85.45.162", 1000, new Point(120,30)))).start(); // MICROSCOPE NODE


   // Non Functioning Base Stations Nodes
  //(new Thread(new MerlListener(nodes, ”ladybug.media.mit.edu”, 8484))).start();
  //(new Thread(new MerlListener(nodes, ”termite.media.mit.edu”, 8484))).start();
  // kitchenNode.drawStripChart(datalist);
```

```
/**
 * @author Manas Mittal
 * Main Display Class
 * Only 1 thread runs here
 *
 */

package edu.media.resenv.clairvoyant;
import java.io.BufferedReader;
import java.io.FileReader;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.StringTokenizer;

import org.eclipse.swt.SWT;
import org.eclipse.swt.events.PaintEvent;
import org.eclipse.swt.events.PaintListener;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.events.SelectionListener;
import org.eclipse.swt.graphics.Color;
import org.eclipse.swt.graphics.Cursor;
import org.eclipse.swt.graphics.Font;
import org.eclipse.swt.graphics.GC;
import org.eclipse.swt.graphics.Image;
import org.eclipse.swt.graphics.Rectangle;
import org.eclipse.swt.graphics.Transform;
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.widgets.Canvas;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Event;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Listener;
import org.eclipse.swt.widgets.Scale;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.ToolBar;
import org.eclipse.swt.widgets.ToolItem;

/**
 * @author Manas Mittal
 * BigMapView.java
 * View Class, Displays all the nodes, and allows for node manipulation.
 *  Parent should call MerlSensorListParser and make nodes
 **/

public class BigMapView implements
Runnable, PaintListener, Listener, SelectionListener {
 LinkedHashMap nodes;
 Image image;
 static Display display;
 Shell shell;
 Canvas canvas;
 GC gc;
 int[][] Rooms;// rooms
 int nRooms = 0; // dc
 static volatile boolean mouseDn = false;
 static volatile Point origin = new Point(0,0);
 static volatile Point endpt = new Point(0,0);
 double scale = (CONSTANT.SCALE);
 static boolean toDraw = true;
 static volatile double avgActivity = 0;
// Label Activity;
// Scale minActScale;
// Scale maxActScale;
// Label minActivity;
// Label maxActivity;
 ToolItem createRule;
 ToolItem createMultiRule;
 ToolItem zoomIn;
 ToolItem zoomOut;
 ToolItem getDetails;
 ToolItem pan;
 Cursor handCursor;
 Font font;
 ArrayList<Rule> rules;
 static int cursorDia = 40;
 static int cursorDiaInner = 24;
 static double thetaOffset = 0;
 static double theta = thetaOffset;

 public BigMapView(Map<String, Node> nodes) {
  // the list passed is already populated with all the nodes
  this.nodes = (LinkedHashMap) nodes; // All the Nodes
  this.rules = new ArrayList<Rule>(); // All the Rules
  //that have to be detected
  this.setLocalizationResolution(CONSTANT.LOCALIZATION_RESOLUTION);
 }

 /**
  * Function : parseRooms
  *  @category : Parses text file corresponding to the location of the roots
  **/
 private void parseRooms() {
  // take the filename and parse it
  this.nRooms = 0;
  this.Rooms = new int[CONSTANT.MAXROOM][];
  int []temp = new int[CONSTANT.MAXPOINTS];
  int yMax = 162;

  try {
   FileReader fr = new FileReader
 ("C:/eclipse/Clairvoyant/src/edu/media/resenv/clairvoyant/mlplan.txt");
   BufferedReader br = new BufferedReader(fr);
   String line;
   while ( (line = br.readLine()) != null) {
    StringTokenizer linet = new StringTokenizer(line, "\n");
    while (linet.hasMoreTokens()) {
     int nPoints = 0;
   StringTokenizer st = new StringTokenizer(linet.nextToken(), " ");
     while(st.hasMoreTokens()) {
      // populate the room
     int v1 = (int) Math.round(Double.valueOf(st.nextToken()));
     int v2 = (int) Math.round(Double.valueOf(st.nextToken()));
     //System.out.println(”nPoints is” + nPoints);
      temp[nPoints++] = v1; // x-Coordinate
      temp[nPoints++] = yMax - v2; // y-coordinate
     }

     temp[nPoints++] = temp[0]; // pad to complete
     temp[nPoints] = temp[1];
    Rooms[nRooms] = new int[nPoints];
    // copy the array
    System.arraycopy(temp, 0, Rooms[nRooms], 0, nPoints);
    nRooms++;
    if(CONSTANT.DEBUG)
     System.out.println("Room "
       + nRooms + " has " + nPoints);
   }
  }
  // Add one more line
  Rooms[nRooms++] = new int[]{0,162,162,162}; // For the Edge
 }
 catch (Exception e) {
  e.printStackTrace();
 }
 if(CONSTANT.DEBUG)
  System.out.println("no of rooms is " + nRooms);
}

static double avg = 0;
static int size = 5;
static double thetaLowPass[] = new double[size];
static int lastEntryIndex = 0;
static Point location = new Point(540,120);


/**
 * Function controls the size of the cursor
 * @param i: 1 —> Maximum Accuracy
 * Default is 2
 */
public void setLocalizationResolution(int accuracy) {
 BigMapView.cursorDia = accuracy * 20;
 BigMapView.cursorDiaInner = accuracy * 12;
}

private double smoothenTheta(double newTheta) {
 avg -= thetaLowPass[lastEntryIndex]/size;
 thetaLowPass[lastEntryIndex] = newTheta;
 lastEntryIndex = (lastEntryIndex+1)%size;
 avg += newTheta/size;
 return avg;
}


private void drawCursor(Shell shell, Display display, GC gc,
  Canvas canvas) {
 // Draw a Circle
 // Call the function Localize to get localization value
 // Start the Serial Port as a separate thread.
 // Call the static function for values for the serial data’

 this.theta =  ((ComPortFuncs.CompassValue[0])/15.0 +
   ((0.1 * ComPortFuncs.CompassValue[1])/256.0)) *  Math.PI * 2;

 this.theta = this.smoothenTheta(this.theta); // smoothen theta out

 // System.out.println(”Theta is ” + this.theta);
 gc.setAntialias(SWT.ON); // Change for N810
 gc.setBackground(Display.getCurrent().getSystemColor(SWT.COLOR_GREEN));
 gc.setAlpha(70);
 gc.fillOval((int)location.getX() - this.cursorDia/2,
   (int) location.getY() - this.cursorDia/2
   , this.cursorDia, this.cursorDia ); // cursor radius
 gc.setBackground(Display.getCurrent().getSystemColor(SWT.COLOR_RED));
 gc.setAlpha(600);
 gc.fillOval((int)location.getX() - this.cursorDiaInner/2,
   (int) location.getY() - this.cursorDiaInner/2
   , this.cursorDiaInner, this.cursorDiaInner);
 // cursor radius : should it be in cosntants instead ?
 /* Make Arrow and Line */
 Point edge1 = new Point ((int) (this.cursorDia/2) * Math.cos(theta),
   (int) ((this.cursorDia/2) * Math.sin(theta)));
 edge1.moveOrigin(location); // edge1 is the center point

 Point offset = new Point ((int) (this.cursorDia/3.4) * Math.cos(theta),
        (int) this.cursorDia/3.4 * Math.sin(theta));
 offset.moveOrigin(location);

 Point edge2 = new
  Point( (-1) * this.cursorDia/6 * Math.sin(theta),
    this.cursorDia/6 * Math.cos(theta));
 edge2.moveOrigin(offset);

 Point edge3 = new Point ( this.cursorDia/6 * Math.sin(theta)
   , -1 * this.cursorDia/6 * Math.cos(theta));
 edge3.moveOrigin(offset);

 int[] points = new int[] { (int)edge2.getX(), (int) edge2.getY(),
       (int)edge1.getX(), (int) edge1.getY(),
       (int)edge3.getX(), (int) edge3.getY() };

 gc.setAlpha(700);
 gc.setLineWidth(1);
 gc.setForeground(Display.getCurrent().getSystemColor(SWT.COLOR_RED));
 gc.setBackground(Display.getCurrent().getSystemColor(SWT.COLOR_RED));
 gc.setLineStyle(SWT.LINE_SOLID);
 gc.drawLine((int) location.getX(), (int) location.getY(),
  (int) (location.getX() + (this.cursorDia/2.1) * Math.cos(theta)),
  (int) (location.getY() + (this.cursorDia/2.1) * Math.sin(theta)));
 // gc.setAlpha(800);
 gc.fillPolygon(points);
 }

public void paintControl(PaintEvent e) {
 this.draw2dMap(shell, display, e.gc, canvas);
 this.drawNodes(display, e.gc, canvas);
 this.drawSelectionBox(shell, display, e.gc, canvas);
 this.drawCursor(shell, display, e.gc, canvas);
}

public void drawSelectionBox(Shell shell, Display display,
  GC gc, Canvas canvas) {
 // System.out.println(”drawSelectionBox called”);
 if(toDraw == true) {

  // gc.setAdvanced(true);
  int p = gc.getAlpha();
  gc.setAlpha(30);
  gc.setBackground(display.getSystemColor(SWT.COLOR_MAGENTA));
  gc.fillRectangle(
    (int)origin.getX(), (int)origin.getY(),
    (int)(endpt.getX() - origin.getX()) ,
    (int)(endpt.getY() - origin.getY()));
  gc.setAlpha(p);
 }
}

public void draw2dMap(Shell shell, Display display, GC gc, Canvas canvas) {
 // draw on the gc
 gc.setForeground(display.getSystemColor(SWT.COLOR_RED));
 gc.setLineStyle(SWT.LINE_SOLID);
 Color clr = new Color(this.display, 164,164,164);
 gc.setLineWidth(3);
 // parse and build an array for this type
 for(int j=0; j<nRooms; j++) {
  int[] room = this.Rooms[j];
  double scale = this.scale;
  int prevx = (int)room[0];
  int prevy = (int) room[1];
  int currx;
  int curry;
  // step 2 each time
  int len = room.length;
  // complete the room;
  int count = 0;
  for (int i = 2; i + 1< len; i = i+2) {
   gc.setForeground(clr);
   currx = (int)room[i];
   curry = (int)room[i+1];
   gc.drawLine( (int) scale * prevx, (int) scale * prevy,
     (int) scale * currx, (int) scale * curry);
   prevx = currx; prevy = curry;
   count++;
  }
 }

 Transform t = new Transform(gc.getDevice());
 t.translate((int) (this.scale/2 * 37f),(int) (this.scale/2 *181f));
 t.rotate(-90f);
 gc.setTransform(t);
 gc.drawText("ELEVATOR", 0, 0);
 t.rotate(90f);
 t.translate((int)(-37f*this.scale/2),(int) (-181f*this.scale/2));
 gc.setTransform(t);

 gc.drawText("ATRIUM", (int) (86 * this.scale/2),
   (int) (200 * this.scale/2));
 gc.drawText("PLW", (int) (10*scale/2) , (int) (212*scale/2) );
 gc.drawText("POND", (int) (175 *scale/2), (int) (200*scale/2));
 gc.drawText("TTT", (int) (251 *scale/2), (int) (290*scale/2));
 gc.drawText("RESENV", (int)(246 *scale/2), (int)(47*scale/2));
 gc.drawText("TMG", (int)(246*scale/2), (int)(83*scale/2));
 gc.drawText("KITCHEN", (int)(263*scale/2), (int)(163*scale/2));
 gc.drawText("OBM", (int)(150*scale/2),(int) (40*scale/2));

}


ToolBar bar;

public void drawImage() {
  // the if is just as a sanity check
  //this.display = new Display();
  this.display = new Display();
  this.handCursor = new Cursor(this.display, SWT.CURSOR_HAND);
  // might have to show this image
  shell = new Shell(display);
  shell.setText("Ubicorder");
  shell.setImage( new Image
(this.display,
  "/src/edu/media/resenv/clairvoyant/icons/Scan.ico"));
  shell.setBackground(Display.getDefault().getSystemColor(SWT.COLOR_WHITE));

  /* Canvas Init */
  canvas = new Canvas(shell, SWT.DOUBLE_BUFFERED);
  canvas.setBackground(display.getSystemColor(SWT.COLOR_LIST_BACKGROUND));
  canvas.addListener(SWT.MouseDown, this);
  /* MouseMove, MouseUp, MouseDown */
  canvas.addListener(SWT.MouseUp, this);
  canvas.addListener(SWT.MouseMove, this);
  canvas.setSize(670,670);
  canvas.setLocation(20,10);
  canvas.addPaintListener(this);
  // this is what will be continuously redrawn


  this.bar = new ToolBar(shell, SWT.VERTICAL|SWT.BORDER);
  // bar.setBackground(new Color(display, 200,20,105));
  //bar.setForeground(new Color(display, 200,50,200));
  bar.setLocation(700,20);
  bar.setSize(120,500);
  bar.setBackground(new Color(display, 250,250,205));

  createRule = new ToolItem(bar, SWT.PUSH);
  createRule.setText("Rule");
  createRule.addListener(SWT.Selection, this);
  createRule.setToolTipText("Create Rule");

  this.createMultiRule = new ToolItem(bar, SWT.PUSH);
  createMultiRule.setText("EDITY");
  createMultiRule.setToolTipText(
    "Define Event Corresponding to Sensor Conditions");
  createMultiRule.addListener(SWT.Selection, this);

  // Zoom and Pan Section
  this.zoomIn = new ToolItem(bar, SWT.PUSH);
  zoomIn.setText("Zoom In");
  zoomIn.setToolTipText("Zoom Into a Region");
  zoomIn.addListener(SWT.Selection, this);

  this.zoomOut= new ToolItem(bar, SWT.PUSH);
  zoomOut.setText("Zoom Out");
  zoomOut.setToolTipText("Zoom out of a Region");
  zoomOut.addListener(SWT.Selection, this);
  // change the corresponding cursor

  this.pan = new ToolItem(bar, SWT.PUSH);
  pan.setText("Pan");
  pan.setToolTipText("Move around the Map");
  pan.addListener(SWT.Selection, this);
  // change the corresponding cursor
  this.getDetails = new ToolItem(bar, SWT.PUSH);
  getDetails.setText("Details");
  getDetails.setToolTipText("Get Details");
  getDetails.addListener(SWT.Selection, this);
  // change the corresponding cursor
  bar.pack();

  // createMultiRule.setSize(70, 30);
  // createMultiRule.setLocation(750, 350);
  this.font = new Font(display, "Arial", 16, SWT.BOLD);

  new Runnable() {
   public void run() {
    canvas.redraw(); // paintControl will redraw As needed
    display.timerExec(100, this);
   }
  }.run();

  shell.setMaximized(true);
  shell.open();
//  shell.pack();
  shell.setMaximized(true);
  try {
   while(!shell.isDisposed()) {
    if(!display.readAndDispatch())
     display.sleep();
   }
  }
  catch (Exception e) {
   System.out.println("Exception : BigMapView");
  }
  gc.dispose();
  canvas.redraw();
  image.dispose();
  display.dispose();
 }

 /*
  * Function to Check if the nodes lie in the given box
  */

 private boolean checkHit(Point p, Point corner1, Point corner2) {
  long minx = Math.min(corner1.getX(), corner2.getX());
  long miny = Math.min(corner1.getY(), corner2.getY());
  long maxx = Math.max(corner1.getX(), corner2.getX());
  long maxy = Math.max(corner1.getY(), corner2.getY());
  long x = (int) (p.getX() * CONSTANT.SCALE);
  long y = (int) (p.getY() * CONSTANT.SCALE);

  if (x > minx && x < maxx && y > miny && y < maxy)
   return true;

  else return false;
 }

 /**
  * @todo This checkNode should only be called when a new Box is drawn
  * @param code
  */

 private void checkNodes(int code) { /*LinkedHashMap */
```

```
 // System.out.println("Check Node Called with code " + code);
 int nodeCount = 0;
 double totalActivity = 0;
 if (2 == code) {
  // return all the nodes within the bounds of the rectangle
  Iterator iter = nodes.keySet().iterator();
  double avgActivity;
  try {
   while(iter.hasNext()) {
    String key = (String) iter.next();
    Node node = (Node) nodes.get(key);
    avgActivity = node.getAverageMerlActivity();
    if(checkHit(node.location, this.origin, this.endpt)) {
     node.selectNode();
     // get the average activity of this node
     // you can also automatically color the complete region
     totalActivity += node.getAverageMerlActivity();
     nodeCount++;
    }
    else {
     node.unselectNode();
    }
   }
  } catch(java.util.ConcurrentModificationException C) {
   System.out.println("Concurrent Exception " +
     ": BigMapView::CheckNode : " + C.toString());
   System.out.println("exiting checknode " +
     "without checking");
   return;
  }
  catch (Exception e) {
   // the nodelist had been modified while iterating
   System.out.println("Exception Issued");
   e.printStackTrace();

   return;
  }

  // Average Node Activity of the selected Region
  if (nodeCount != 0)
   this.avgActivity = totalActivity / nodeCount;
  else
   this.avgActivity = 0.0;
 } // code == 1
} // end function

public void drawNodes(Display display, GC gc, Canvas canvas) {
  // TODO: should this list be syncronized ?
 Iterator iter = nodes.keySet().iterator();
 // draw each of the nodes
 int count = 0;
 while(iter.hasNext()) {
  String key = (String) iter.next();
  Node node = (Node) nodes.get(key);
  // System.out.println("Name is " + node.name);
  node.drawNode(gc, display, canvas);
  count++;
 }
}

/* VizKeepAlive - Visualize the keepalive signal from a node */
```

```
 public Boolean VizKeepalive (String nodename) {
  Node p;
  if((p = (Node) this.nodes.get("nodename")) != null) {
   return(this.vizKeepAlive(p));
  }
  else return false;
 }

 public Boolean vizKeepAlive(Node node) {
  return true;
 }

 public void vizMotion(Node node) {
  // Hello
  // World

 }

 public void run() {
  this.parseRooms(); // doesn't loop continuously
  this.drawImage();
 }

 public void setNodes(LinkedHashMap nodes) {
  this.nodes = nodes;
 }

 public void handleEvent(Event e) {

  if(e.widget == zoomIn) {
   Node n = (Node) this.nodes.get("01001034"); // sample MERL Node
   BrowseInfoShell browseInfoShell = new BrowseInfoShell(this.shell);
   browseInfoShell.setNode(n);
  }

  if(e.widget == zoomOut) {

  }

  if(e.widget == pan) {
   this.canvas.setCursor(this.handCursor);

  }

  if(e.widget == this.getDetails) {

  }

  if(e.widget == createRule) {
   System.out.println("Button Pressed");
   try {
    Rule rule = new Rule(this.shell);
    rules.add(rule); // this is a list of current rules
    // deleting rules will equal removing it from this list
         // rule has a constructor
   } catch (Exception e1) {
    e1.printStackTrace();
   }
   return;
  }


  if(e.widget == createMultiRule) {
   System.out.println("MultiRule Button Pressed");
   try {
    Node n = (Node) this.nodes.get("01001034"); // sample MERL Node
    if (n == null) {
     System.out.println( " Error - Node Does Not Exist" );
    }

    MultiRuleView multiRuleView = new MultiRuleView(n, this.shell);
   } catch (Exception e1) {
    System.out.println("Check if Node is inited");
    e1.printStackTrace();
   }
   return;
  }

  if(e.type == SWT.MouseUp) {
   if(CONSTANT.DEBUG)
    System.out.println("MouseUp " +
      "with " + e.x + "," + e.y + " as the points");
   BigMapView.endpt.setXY(e.x, e.y);
   this.mouseDn = false;
   checkNodes(1);
  }

  else if(e.type == SWT.MouseDown) {
   if(CONSTANT.DEBUG)
    System.out.println("MouseDown with "
      + e.x + "," + e.y + " as the points");
   BigMapView.mouseDn = true;
   BigMapView.origin.setXY(e.x, e.y);
  }

  else if(BigMapView.mouseDn == true && e.type == SWT.MouseMove) {
    BigMapView.endpt.setXY(e.x, e.y);
    checkNodes(1);
  }
 }

 public void widgetDefaultSelected(SelectionEvent arg0) {
  // TODO Auto-generated method stub
  /// this.checkNodes(1);
 }

 public void widgetSelected(SelectionEvent arg0) {
  /// this.checkNodes(1);
 }
}
```

```
package edu.media.resenv.clairvoyant;
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Observable;
import java.util.Random;

import org.eclipse.swt.SWT;
import org.eclipse.swt.graphics.Color;
import org.eclipse.swt.graphics.Cursor;
import org.eclipse.swt.graphics.GC;
import org.eclipse.swt.graphics.Rectangle;
import org.eclipse.swt.widgets.Canvas;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Event;
import org.eclipse.swt.widgets.Listener;

/** @perfcomment
 *
 * @author Manas Mittal
 *
 */

public class Node extends Observable implements Listener {
 public String name;
 HashMap<String, Sensor> sensors = new HashMap<String, Sensor>();
 HashMap<String, Tag> tags = new HashMap<String,Tag>();
 Point location;    // is not scaled
 int count;
 Color boxclr;
 public volatile Long lastTime = new Long(0);
 Rectangle scaledIcon;
 volatile char type;
 volatile int selectColorCode = -1;
 static String merl = CONSTANT.merl;

 double features[] = new double[1]; // Compute all the features
            // Then when you have to select nodes with the given feature,
            // Just walk down the node list and select them.
 // do feature computation
    public Box box; // may or may not be used
 Display display;
 Canvas canvas;
 GC gc;
 volatile int side = 1;
 volatile Color nodeColor = null;
// volatile boolean empty = true;
// private boolean listeners = false;
 double scale;
 private boolean mouseDown = false;
 Color selectColor = null;

 public Node(String name, Point location, char type) {
  this.type = type;
  this.name = name;
  this.location = location;

  if (this.type == CONSTANT.TYPE_SPINNERWALL) {
   sensors.put("Light", new Sensor(name + "- Light", 150, 1600)); //
   sensors.get("Light").setAvgWindow(5);
   sensors.put("Sound", new Sensor(name + "- Sound", 200, 30000)); // additional sensors
   sensors.get("Sound").setAvgWindow(20);
   sensors.put("Humidity", new Sensor(name + "- Humidity",0,100));
   sensors.get("Humidity").setHumidity(true);
   sensors.put("Temperature", new Sensor(name + "- Temperature",0,8000));
   sensors.put("Movement", new Sensor(name + "- Movement",0,1));
   this.scale = CONSTANT.SCALE;
   this.scaledIcon = new Rectangle((int) (scale* location.getX() - side),
     (int) ((scale* location.getY()) - side) ,
     (int) 4 * this.side, (int) 4 * this.side);
  }

  if (this.type == CONSTANT.TYPE_MERL) {
   this.scale = CONSTANT.SCALE;
   sensors.put(merl, new Sensor(CONSTANT.TYPE_MERL, name + "-Movement"));
   this.scale = CONSTANT.SCALE;

   this.scaledIcon = new Rectangle((int) (scale* location.getX() - 4*side),
           (int) ((scale* location.getY()) - 4*side) ,
           (int) 8 * this.side, (int) 8 * this.side);
  }
 }

 public boolean drawNode(GC gc, Display display, Canvas canvas, double scale) { // Called by Map
  if (this.selectColor == null) {
   this.selectColor = display.getSystemColor(SWT.COLOR_MAGENTA);
  }

  if (this.selectColorCode != -1) {
   this.selectColor = display.getSystemColor(SWT.COLOR_MAGENTA);
  }

  this.gc = gc;

  if(this.canvas != canvas) {
   this.canvas = canvas;
   // expensive, adding listeners
   this.canvas.addListener(SWT.MouseDown, this);
   this.canvas.addListener(SWT.MouseMove, this);
  }

  if (this.scale != scale) {
   this.scaledIcon.x = (int) (scale * this.location.getX() - side);
   this.scaledIcon.y = (int) (scale * this.location.getY()- side);
  }

  this.display = display; // ysed in simple drawNode
  this.drawNode(gc,  display.getSystemColor(SWT.COLOR_GREEN), scale);
  return true;
 }


 public boolean drawNode(GC gc, Display display, Canvas canvas) {
  return this.drawNode(gc, display, canvas, CONSTANT.SCALE);
 }

 volatile boolean selected = false;
 public void selectNode() {
//  System.out.println(" Node.java : Select Node Called");
  this.selected = true;
  this.update();
  // inform Node Called
  // how to get signal to the display thread to redraw the canvas ?
 }

 public void selectNode(int colorCode) {
  System.out.println(" Node.java : Select Node Called with ColorCode " + colorCode);
  this.selectColorCode = colorCode;
  this.selected = true;
  this.update();
  // how to signal to the display thread to redraw the canvas
 }

 public void unselectNode() {
  if (this.selected) {
  }
  this.selected = false;
  this.update();
 }

 public boolean isSelected() {
  return this.selected;
 }

 private synchronized void update() {
  // also decrease the size if > 2 seconds since last increase
  /* Ornate mechanism
   * to ensure that the size only gets decremented
   * after 1000 seconds */
  long t = System.currentTimeMillis();
  if (this.side > 1) {
   Sensor sensor = sensors.get(merl);
   this.side--;
   if((t - sensor.getLastActiveTime()) > CONSTANT.SHRINKTIMEMILISECOND) {
//    this.countPings--;
    this.lastTime = t;
   }
  }
  if (this.countPings == 0) {
   if(display != null) {
    this.nodeColor = new Color(Display.getCurrent(), 56,56,56);
//    this.canvas.redraw();
   }
  }
  // this.canvas.redraw(); -- There is a stream that constantly redraws the nodes
 }

 double maxValue = 0;
 Color gray = null;
 Color red = null;
 Color mustard = null;
 Color bigBoxColor = null;
 volatile boolean isBoxed = false;

 public void setBoxed(boolean b) {
  this.isBoxed = b;
  this.bigBoxColor = this.red;
 }

 public void setBoxed(boolean b, Color clr) {
  this.isBoxed = b;
  this.bigBoxColor = clr;
 }
```

```
 Color alphaBoxColor = null;
 boolean alphaBox = false;
 public void setAlphaBox(boolean b) {
  this.alphaBox = b;
  this.alphaBoxColor = this.selectColor;
 }

// double theta = Math.PI/4;
// double thetaInc = Math.PI/32;

 public synchronized boolean drawNode(GC gc, Color clr, double scale) {
  // actual drawing happens here
//  this.update();
  if (this.selectColor == null) {
   this.selectColor = (Display.getCurrent().getSystemColor(SWT.COLOR_MAGENTA));
  }

  if (this.gray == null || this.red == null || this.mustard == null) {
   this.gray = new Color(this.display, 164,164,164);
   this.red = Display.getCurrent().getSystemColor(SWT.COLOR_RED);
   this.mustard = new Color(this.display, 255,153,51);
  }

  if (this.selectColorCode != -1) {
   this.selectColor = Display.getCurrent().getSystemColor(this.selectColorCode);
  }

  if (this.nodeColor == null) {
   nodeColor = new Color(Display.getCurrent(), 56,56,56);
  }

  this.scale = scale;
  // ignoring scale as scaledRectangle has already been calculated.
  int l = this.side;
  int x = (int) (location.getX()* scale);
  int y = (int) ((int) (location.getY() * scale ));

  // Sound
  // Temperature
  Sensor sound = this.sensors.get("Sound");
  if(sound!=null && gc.getAdvanced()) { // This sensor actually exists
   gc.setAlpha(70);
   gc.setForeground(gray);


   double rSmall = 1;
   double r = (this.sensors.get("Sound").getLastValue()/4000);

   if (CONSTANT.DEMO_MODE && r == 0) { // create synthetic visualizati
    Random random = new Random();
    r = 1 +  random.nextDouble();
   }

   r += rSmall;

   gc.fillOval( (int) (x  - 2*l*scale*r) , (int) (y - 2*l*scale*r),
     (int) (4*l * scale * (r)) , (int) (4*l * scale * (r)));
   gc.drawOval( (int) (x  - 2*l*scale*r) , (int) (y - 2*l*scale*r),
     (int) (4*l * scale * (r)) , (int) (4*l * scale * (r)));
   gc.setAlpha(1000);
}

if(gc.getAdvanced()) {
 Sensor light = this.sensors.get("Light");
 if (light != null) {
  gc.setAlpha(1000); // will only work if gc.setAlpha is turned on
  gc.setForeground(mustard);
  gc.setLineWidth(3);
  long r = Math.round((this.sensors.get("Light").getLastValue()/150));
  if (CONSTANT.DEMO_MODE && r == 0) { // create synthetic visualizati
   Random rdm = new Random();
   r = Math.round(1 +  3 * rdm.nextDouble());
  }

  long rSmall = Math.round(scale * 4);
  r = r + (int) rSmall + Math.round(scale);

  double cosTheta = 0.707;
  double sinTheta = 0.707;

  double rSmallDelX = rSmall * cosTheta;
  double rSmallDelY = rSmall * sinTheta;
  double rBigDelX = r * cosTheta;
  double rBigDelY = r * sinTheta;

  gc.setLineWidth(2);
  gc.setLineStyle(SWT.LINE_DOT);
  gc.drawLine( (int) (x + rSmallDelX),(int)(y + (int)rSmallDelY)
    , (int) (x + rBigDelX),(int) (y + rBigDelY));
  gc.drawLine( (int) (x + rSmallDelX), (int) (y - (int)rSmallDelY)
    , (int) (x + rBigDelX),(int) (y - rBigDelY));
  gc.drawLine( (int) (x - rSmallDelX),(int) (y + (int)rSmallDelY),
    (int) (x - rBigDelX),(int) (y + rBigDelY));
  gc.drawLine( (int) (x - rSmallDelX),(int) (y - (int)rSmallDelY),
    (int) (x - rBigDelX),(int) (y - rBigDelY));

  gc.drawLine( (int) (x),(int) (y + rSmall), (int) (x),(int) (y + r));
  gc.drawLine( (int) (x - rSmall),(int) (y), (int) (x - r),(int) (y));
  gc.drawLine( (int) (x + rSmall),(int) (y), (int) (x + r),(int) (y ));
  gc.drawLine( (int) (x),(int) (y - rSmall), (int) (x),(int) (y - r));
 }
}

if (this.isBoxed) {
 gc.setAlpha(1000);
 gc.setLineStyle(SWT.LINE_SOLID);
 gc.setForeground(this.bigBoxColor);
 gc.drawRectangle((int) (x - scale*5), (int) (y - scale * 5),
   (int) (10*scale), (int) (10*scale));
}

if (this.alphaBox) {
 gc.setAlpha(300);
 gc.setBackground(this.alphaBoxColor);
 gc.fillRectangle((int) (x - scale*5), (int) (y - scale * 5),
   (int) (10*scale), (int) (10*scale));
}

// Draw the node itself
gc.setForeground(clr);
gc.setAntialias(SWT.ON);
```

```
 if (this.type == CONSTANT.TYPE_SPINNERWALL) {
  if (! this.selected) {
   gc.setBackground(nodeColor);
  }
  else {
   gc.setBackground(selectColor);
   gc.setForeground(display.getSystemColor(SWT.COLOR_CYAN));
  }
  gc.setAlpha(1000);
  gc.fillOval( (int) (x  - 2*l*scale) , (int) (y - 2*l*scale),
    (int) (4*l * scale) , (int) (4*l * scale));
  this.scaledIcon = new Rectangle ((int) (x  - 2*l*scale) ,
    (int) (y - 2*l*scale), (int) (4*l * scale) ,
    (int) (4*l * scale));
 }

 if (this.type == CONSTANT.TYPE_MERL) {
  if (!this.selected) {
   gc.setBackground(nodeColor);
   gc.setForeground(nodeColor);
  }
  if (this.selected) {
   gc.setBackground(selectColor);
   gc.setForeground(display.getSystemColor(SWT.COLOR_CYAN));
  }
  this.scaledIcon = new Rectangle ((int) (x  - l*scale) ,
    (int) (y - l*scale), (int) (2*l * scale) ,
    (int) (2*l * scale));
  gc.fillRectangle((int) (x  - l*scale) , (int) (y - l*scale),
    (int) (2*l * scale) , (int) (2*l * scale));
 }
 return true;
}

public HashMap<String, Sensor> getSensors() {
 return this.sensors;
}

public synchronized void addSpinnerData(double[] data, long seqno) {
 // seqno may be useful in the future
 if(this.type == CONSTANT.TYPE_SPINNERWALL) {
  Sensor s = sensors.get("Light");
  s.addData(data[0]);

  s = sensors.get("Sound");
  s.addData(data[1]);

  s = sensors.get("Humidity");
  s.addData(data[2]);

  s = sensors.get("Temperature");
  s.addData(data[3]);

  s = sensors.get("Movement");
  s.addData(data[4]);
 }
}

volatile int countPings = 0;
public synchronized void addMerlPing(String [] data) {
 this.lastTime = System.currentTimeMillis();
```

```
  this.setChanged(); // notification will work now
  this.notifyObservers();
  // Tell the Observers that this event happened. !!!

  countPings++;
  if (this.countPings > 0) {
   try {
    this.nodeColor = new Color(Display.getCurrent(), 112,156,156);
    // Mark as greener
   } catch(Exception e) {
    e.printStackTrace();
    }
  }

  if (this.side <= 1)  {
   this.side++; // this will increase the size of an edge.
   this.scaledIcon.x = (int) (scale * this.location.getX() - side);
   this.scaledIcon.y = (int) (scale * this.location.getY()- side);
  }

  // Start a new thread that will ”decrease the size of the merl node”
  // Make it shrink back !
  Thread shrinkNode = new Thread() {
   public void run() {
    try {
     Thread.sleep(CONSTANT.SHRINKTIMEMILISECOND);
     update();
    } catch (InterruptedException e) {
     // TODO Auto-generated catch block
     e.printStackTrace();
    }
   }
  };
  shrinkNode.start();


  if(CONSTANT.DEBUG) {
  System.out.println("SIZE OF DISPLAY NODE updated");
  System.out.println("CHANGED SIZE of node " + this.getCount());
  }

  Sensor sensor = sensors.get(merl);
  /* Add Ping logs that sensor received a entry */
  sensor.addPing();
  /* data[4] is the time stamp */
  if(data[4] != null) {
   List <String> p = sensor.strList;
   p.add(data[4]);
   /* Change color for 2 seconds / trigger an animation */
  }
 }

 public synchronized  void pushMerlData(LinkedHashMap<String, String> data) {

 }
 public int getCount() {
  return count;
 }
 public double getAverageMerlActivity() {

  return 2.0* (this.side - 1.0); // HACK
```

```
//  Sensor sensor = sensors.get(”merl”);
//  return sensor.getAverageActivity();
 }

 public void setCount(int count) {
  this.count = count;
  // System.out.println(”Count is set as” + this.count);
 }

 boolean marked = false; // local selected indicator
 boolean reallySelected = false;
 // Marked is only used in this function to ”light up” the node
 // Choose a better method, perhaps enclose in a composite
 public void handleEvent(Event e) {

  if(e.type == SWT.MouseMove) {
   if (this.scaledIcon.contains(e.x, e.y)) {
    if (this.marked == false) {
     this.canvas.setCursor(new Cursor(Display.getCurrent(),
       SWT.CURSOR_HAND));
     this.selectNode();
     this.marked = true;
    }
   } else if(this.marked){
    this.canvas.setCursor(null);
    this.marked = false;
    if(! this.reallySelected) {
     this.unselectNode();
     this.canvas.redraw();
    }
   }
  }

  else if (e.type == SWT.MouseDown) {
   // System.out.println(”Double Click at ” + e.x + ”,” + e.y);
   if (this.scaledIcon.contains(e.x, e.y)) {
    this.reallySelected = true;
    this.selectNode();
    this.setChanged();
    this.notifyObservers(this); // Explicit Instruction
    this.canvas.redraw();
   } else {
    // clicked somewhere else,
    // mark as not selected
    this.reallySelected = false;
    this.unselectNode();
    // Notify all Listeners about the Unselection
   }
  }
 }

}
```

```
package edu.media.resenv.clairvoyant;
import java.util.ArrayList;
import java.util.List;
import java.util.Observable;
import java.util.Observer;
import java.util.concurrent.ArrayBlockingQueue;

/*
 * Stores Data for each Sensor
 * Registers Listeners, so that visualization and rules can be triggered.
 */

public class Sensor extends Observable implements SensorInterface {
// public enum OutputType {Analog, Binary};
 // public enum Type {Light, Temp, PIR, Vibration, Mic, MERL};
 // What does it sense
  // List<Data> data = new ArrayList();
 final int size = 100;
 ArrayList<Double> bufferList = new ArrayList<Double>();
 ArrayList<Sample> sampleList = new ArrayList<Sample>();

 volatile double buffer[] = new double[size];
 // volatile leads to a big perf hit, I would assume. Check!
 volatile long seqnos[] = new long[size];
 // volatile leads to a big perf hit, I would assume
 volatile List<Long> time = new ArrayList<Long>();
 // 0 to TIMELENGTHMAX - 1 values
 volatile int lastTime = -1;
 private volatile long lastActiveTime = -1 ;
 private volatile boolean changedRecently = false;
 List<String> strList = new ArrayList <String>();
 int type = -1;
 int bufferIndex = 0;
 boolean bufferFilled = false;
 // remembers if the buffer of 50 elements is filled up or not
 volatile int streamListeners = 0;
 final static int pulseWidth_ms = 2000;
 final static int updateRate_ms = 500;
 private Thread merlUpdateThread = null;
 int maxValue = 1;
 int minValue = 0;
 String name = "(Default) Sensor Name";
 private boolean humidity = false;
 int avgWindowSize = 1;
 ArrayBlockingQueue<Double> smoothBufferList =
  new ArrayBlockingQueue<Double>(avgWindowSize);

 double runningAvg = 0;

 public void setHumidity(boolean yesno) {
  this.humidity = yesno;
 }

 public void setAvgWindow(int w) {
  this.avgWindowSize = w;
  this.sum = 0;
  this.count = 0;
  this.smoothBufferList = new
  ArrayBlockingQueue<Double>(this.avgWindowSize);
 }

 public Sensor(int type, String name) {
  this.name = name;
  this.type = type;
//  this.bufferList.add(0.0);
//  this.sampleList.add(new Sample(0,System.currentTimeMillis()));
  // do things to initialize this type
  switch(type) {
  case CONSTANT.TYPE_MERL :
//   System.out.println("A Merl Type of Sensor Registered");
   // this means that AddPing will be called
   this.merlInit();
   // merlUpdateThread converts the
   // Binary, occasionally occuring YES/NO values to
   // a data stream, after the startMerlStreaming function is called.
   // This is used for displaying the sensor Stream

   break;
  default:
   this.merlInit();
   break;
   // how to call the default method.
  }
 }


 public int getType() {
  return this.type;
 }

 public Sensor() {
  this.merlInit(); // Bad Naming, should really be some form of init
 }

 public Sensor(String name, int min, int max) {
  this.name = name;
  this.minValue = min;
  this.maxValue = max;
  this.merlInit();
 }

 public Double[] getBuffer() {
  return (Double[]) bufferList.toArray();
 }

 private volatile double lastValue = 0;

 public synchronized double getLastValue() {
  return this.lastValue;
 }
 volatile double sum = 0;
 volatile int count = 0;
 public void addData(double newData) {
  // Called by Add Spinner Data, and also by the MerlUpdateThread
  if (this.avgWindowSize != 1) {
   if (this.smoothBufferList.remainingCapacity() != 0) {
    this.smoothBufferList.add(newData);
    sum += newData;
    count++;
    return; // fill it up
   }
   else {
    sum -= this.smoothBufferList.remove();
    this.smoothBufferList.add(newData);
    sum += newData;
    newData = sum/count;
   }
  }

  if(this.humidity) {
   // Do the humidity computation
   newData =  (int) Math.round(-4.0 +
     (double) (0.0405 * newData) -
     (2.8e-6 * (double)newData
      * (double)newData));
  }
  bufferList.add(newData);
  // Extra Overhead, for now, all data is pushed into this.
  this.lastValue = newData;
  Sample newSample = new Sample(newData, System.currentTimeMillis());
  sampleList.add(newSample); // final: add the timeStamp + Data
  if (this.sampleBufferList.remainingCapacity() == 0) {
   this.sampleBufferList.remove();
  }
  this.sampleBufferList.add(newSample);
  // inform all the SimpleRules of this new data
  this.setChanged();
  this.notifyObservers(newSample);
  // Tell the SimpleRules that the event happened
 }



 public ArrayList getCompleteSensorData() {
  return this.bufferList;
 }

 private void merlInit() {
  for (int i = 0; i < CONSTANT.TIMELENGTHMAX; i++) {
   time.add(new Long(CONSTANT.LNGNOTVALID)); // initialize to values
  }
 }

 public synchronized void startStreaming() {
  System.out.println("Start Merl Streaming Called");
  if (this.type == CONSTANT.TYPE_MERL) {
   // Should be singleton
   // New thread, updates once every second
   // thread checks if this.cha
   if(this.streamListeners == 0) {
    // Only start if streamListeners is currently = 0
//    System.out.println("Started Streaming");
    if( (this.streamListeners++ == 0)) {
     merlUpdateThread = new Thread() {
      public void run() {
       System.out.println(
        "MerlUpdateThread IS RUNNING ***"
         + this.toString());
       while(streamListeners > 0) {
         if(changedRecently == true) {
          // set true by AddPing
          // Get the current time,
          //and the last ping time
          // if differece <
          if (
           (System.currentTimeMillis()
           - lastActiveTime) < pulseWidth_ms){
           addData(1D);
      // add 1 to the Array or whatever (RingBuffer perhaps)
          }
          else {
           changedRecently = false;
           addData(0D);
          }
         }
         else {
          addData(0D);
         }
        try {
         Thread.sleep(updateRate_ms);
        } catch (InterruptedException e) {
         System.out.println(
           "Error in merlUpdateThread");
         e.printStackTrace();
        }
       } // end while
      } // end run
     };
     merlUpdateThread.start();
    }
   } // end if condition
  }
 }


 public synchronized void stopStreaming() {
  System.out.println("Stopped Merl Streaming for");
  if (this.type == CONSTANT.TYPE_MERL) {
   --this.streamListeners;
   if(this.streamListeners == 0) {
    System.out.println("STOP THE THREAD !!!");
   }
   if(streamListeners < 0) { // when
 //streamListeners become equal to 0, the thread will stop, automatically
    System.out.println("" +
      "ERROR stopMerlStreaming" +
      " called for more times " +
      "than start merlStreaming");
    this.streamListeners = 0;
   }
  }
 }

 public void addPing() {
  // add the time stamp
//  System.out.println("addPing Called");
  Long t = new Long(System.currentTimeMillis());
  lastActiveTime = t;
  lastTime = (lastTime + 1) % CONSTANT.TIMELENGTHMAX ;
  time.set( lastTime, t);
  this.changedRecently = true;
//  this.setChanged(); // Why ?
//  this.notifyObservers(t); // Notify the time of the last ping
 }
 /**
  *
  * @return : Returns the number of Average activations
  * per second, based on the last CONSTANT.TIMELENGTHMAX Moving Window
  *
  */

 public double getAverageActivity() {
  // Average time between beam breaks
  int oldest = 0;
  int entries = 0;
  long timeold =time.indexOf(CONSTANT.LNGNOTVALID);
  // timeold will be −1 if all entries are filled up
  // 0 if the no entry is there

  if(timeold == 0 || timeold == 1) {
   // no entry exists,
   //or only 1 entry exists.
   // So Determining the average is difficult (MeaningLess)
   return 0.0;
  }

  if ( timeold == -1) { // all entries are filled up
   oldest = (lastTime + 1) % CONSTANT.TIMELENGTHMAX;
   entries = CONSTANT.TIMELENGTHMAX;
  }
  else if (timeold > 1) {
   oldest = 0;
   entries = lastTime;
  }
  return entries;
  //return (entries / ((time.get(lastTime) − time.get(oldest))/1000));
 }

 /** Add Special Observer does the following :
  * Ensures that the artifical sensor data
  *  stream keeps running if this sensor is a MERL Sensors
  *  and if some Rule is watching
  * Must be called deleted by calling deleteSpecialObserver
  * @param O: The Observer, in this case, simpleRule
  */
 public void addSpecialObserver(Observer O) { // Called by SimpleRule
  if (this.type == CONSTANT.TYPE_MERL) {
   // Make sure that the MerlUpdateThread keeps running
   this.startStreaming();
  }
  this.addObserver(O);
 }

 public void deleteSpecialObserver(Observer O) {
  this.deleteObserver(O);
  if(this.type == CONSTANT.TYPE_MERL) {
   this.stopStreaming();
  }
 }

 public long getLastActiveTime() {
  if(lastActiveTime > 0) {
   return lastActiveTime;
  } else {
   return CONSTANT.LNGNOTVALID;// this has never been active
  }
 }

 public int getMaxValue() {
  return maxValue;
 }

 public int getMinValue() {
  return minValue;
 }

 public void setMinValue(int minValue) {
  this.minValue = minValue;
 }

 public void setMaxValue(int maxValue) {
  this.maxValue = maxValue;
 }

 public void setMaxMin(Double max, Double min) {
  this.minValue = (int) Math.floor(min);
  this.maxValue = (int) Math.ceil(max);
 }

 public double getMax() {
  return this.getMaxValue();
 }

 public double getMin() {
  return this.getMinValue();
 }
 ArrayBlockingQueue<Sample> sampleBufferList =
  new ArrayBlockingQueue<Sample>(CONSTANT.WINDOW_SIZE);


 public Sample[] getSampleBuffer() {
  Sample[] s = new Sample[CONSTANT.WINDOW_SIZE];
  return (Sample[]) this.sampleBufferList.toArray(new Sample[0]);

  // MM
  // TODO Auto−generated method stub
//  return null;
 }

 public String getName() {
  return name;
 }

 public void setName(String name) {
  this.name = name;
 }


}
```

```
package edu.media.resenv.clairvoyant;

public interface SensorInterface {
// void addData(double newData);
 void startStreaming();
 void stopStreaming();
 Double[] getBuffer();
 Sample[] getSampleBuffer();
 void setMaxMin(Double max, Double min);
 double getMin();
 double getMax();
 void setName(String name);
 String getName();
}
```

```
package edu.media.resenv.clairvoyant;

import java.net.MalformedURLException;
import java.util.ArrayList;
import java.util.Observable;
import java.util.Observer;
import java.util.Vector;
import java.util.concurrent.ArrayBlockingQueue;

import org.eclipse.swt.SWT;

/**
 * @author Manas Mittal
 * This is really like a sensor.
 */
public class SimpleRule extends Observable implements Observer,SensorInterface{
 // Observes simple sensor data
 // Observable for complexRule
 // complexRule will inherit this SimpleRule
 private Thread updateThread = null;
 protected double threshUpper;
 protected double threshLower;
 Sensor sensor = null;
 Node  node = null;
 int filterCount = 0;
 private ArrayList<Filter> filters; // Filter is an abstract class
 private int actionCode = 0; // 0 means none
 java.applet.AudioClip audioClip = null;
 protected volatile long lastTrueTime = System.currentTimeMillis()
              - pulseWidth_ms;
 private static final  long pulseWidth_ms = 200;
 private static final long updateRate_ms = CONSTANT.UPDATE_RATE;
 int minValue = 0;
 int maxValue = 1;
 private int slack = 0;
 private int zeroDelta = 4;
 protected String name = "Rule Being Created";

 public void setName(String name) {
  this.name = name;
 }

 public String getName() {
  return this.name;
 }

 public void setSlack(int slack) {
  this.slack = slack;
 }

 public void setZeroDelta(int zeroDelta) {
  this.zeroDelta = zeroDelta;
 }

 public void setMaxMin(Double max, Double min) {
  this.minValue = (int) Math.floor(min);
  this.maxValue = (int) Math.ceil(max);
 }

 public double getMax() {
  return (double) this.maxValue;
 }

 public double getMin() {
  return (double) this.minValue;
 }

 public SimpleRule(ArrayList<ComponentRuleDesc> rules) {
  // used only for compound rules.
 }


 public SimpleRule( Node N,
      Sensor sensor,
      double threshLower, double threshUpper) {
  System.out.println("Simple Rule Created with LowerThresh "
    + threshLower + " UpperThresh " + threshUpper);
  // Thresholds Must be scaled to sensor signal space
  this.threshUpper = threshUpper;
  this.threshLower = threshLower;
  this.sensor = sensor;
  this.node = N;
  // Note : The ThreshUpper and the
  // ThreshLower are in Sensor Space (not in co-ordinate space)
  // I probably don't even need the node
  // Register Listeners
  // Remember to Deregister Listeners
  this.filters = new ArrayList<Filter>(); // Filters
  sensor.addSpecialObserver(this);
  // Add Observer to sensor, this
  // starts streaming if node is notcontinuous
  System.out.println("SimpleRule.java : Listener Added to Given Sensor");
 }

 ArrayBlockingQueue<Double> bufferList =
     new ArrayBlockingQueue<Double>(CONSTANT.WINDOW_SIZE);
 ArrayBlockingQueue<Sample> sampleBufferList =
     new ArrayBlockingQueue<Sample>(CONSTANT.WINDOW_SIZE);

 // NOTE THAT THIS IS PRIVATE
 private volatile Sample lastSample = null;
 private void addSample(Sample sample) {
  if (sampleBufferList.size() == CONSTANT.WINDOW_SIZE) {
   sampleBufferList.remove();
  }
  sampleBufferList.add(sample); // Push the time generation to later
  this.lastSample = new Sample(sample.data, sample.timeStamp);
 }

 private synchronized void addData(double data) { // Called by streaming
  if(bufferList.size() == CONSTANT.WINDOW_SIZE) {
   bufferList.remove();
   // remove the oldest element -- removes the oldest element by default
  }
  bufferList.add(data);
  // Add to the Data Buffer
 }

 public Double[] getBuffer() {
  // Called by the StripChart, and should also be
  // called by the complex rule that makes up these simple rules
  // this is a very expensive operation
 // Should be called only once, initially, that is.
 Double[] v = new Double[CONSTANT.WINDOW_SIZE];
 // create an array of the guys
 v = (Double[]) this.bufferList.toArray(new Double[0]);
 return v;
            // Perf Hit
            // Might Need Speedup
}

public Sample[] getSampleBuffer() {
 Sample[] s = new Sample[CONSTANT.WINDOW_SIZE];
 s = (Sample[]) this.sampleBufferList.toArray(new Sample[0]);
 return s;
}



// There will be a class Filter
public int addFilter(int position, int Type, double param) {
 if (Type == CONSTANT.FILTER_SMOOTHEN) {
  Filter f = new smoothFilter((int) param);
  filters.add(f);
  // take the average etc
 }
 return 0;
}

static volatile boolean playing = false;

public void setActionCode (int actionCode) {
 System.out.println("Setting Action Code" + actionCode + " for" + this.name);
 this.actionCode = actionCode;
 if (actionCode == CONSTANT.ACTION_DING) {
  try {
   System.out.println("Setting SOUND Action Code" + actionCode);
   this.audioClip = java.applet.Applet.newAudioClip
    (new java.net.URL("file:/c:/WINDOWS/Media/ding.wav"));
   // change for Nokia N810
  } catch (MalformedURLException e) {
   e.printStackTrace();
  }
 }
}

volatile int streamListeners = 0;
public void startStreaming() {
 System.out.println("SimpleRule: StartStreaming called");
 if(streamListeners++ == 0) {
  this.updateThread = new Thread() {
   public void run() {
    while (true) {
     if( (System.currentTimeMillis() - lastTrueTime)
       < pulseWidth_ms) {
      addData(1D);
     }
     else {
      addData(0D);
     }
     try {Thread.sleep(updateRate_ms);}
     catch(Exception e) {
      e.printStackTrace();
     } // end catch
    } // end while loop
   } // end run
  };
  this.updateThread.start();
  // start the streaming process
 }
}

public void stopStreaming() {
 if(--streamListeners < 0) {
  System.out.println("SimpleRule.java :
      ERROR: stopStreaming called more times that startStreaming");
  streamListeners = 0;
 }
}
int time = 0;
synchronized protected boolean checkRule(Sample newSample
  , Observable sensor) {// will be changed by the other guy
 double newData = newSample.data;
 // This might be useful later
 if ((Double) newData <= this.threshUpper
   && (Double) newData >= this.threshLower) {
  this.lastTrueTime = System.currentTimeMillis();
  // The rule was true here
  // Add this in the local buffer here
  return true;
 }
 return false;
}

public synchronized Sample getLastSample() {
 return this.lastSample;
}

Sample sample = null;

public void update( Observable sensor, Object newData) {
 // Called by Sensor, because of addSpecialObserver
 boolean result = false;
 if (newData instanceof Sample) {
  if ((Sample) newData == null) {
   return;
  }
  if (this.checkRule((Sample) newData, sensor)) {
   result = true;
  } else {
   result = false;
  }

  if (result == true) {
   sample = new Sample (1D, ((Sample)newData).timeStamp);
  } else {
   sample = new Sample (0D, ((Sample)newData).timeStamp);
  }
  this.addData(sample.data);
  // So that it can be visualized by the StripChart's Buffer
  this.addSample(sample); // Add to the local buffer
 }

 if (result == true) { // Perform the corresponding action
//   System.out.println(this.name + " Some rule is true");                              }
   switch(this.actionCode) {                                                         }
   case CONSTANT.ACTION_NODEBOXED:
    if (this.node != null) {
     this.node.setBoxed(true);
    } else {
     System.out.println("Rule Not Linked to a Single Node");
    }
     break;
   case CONSTANT.ACTION_NODECOLOR:
    if (this.node != null) {
     this.node.setAlphaBox(true);
//      this.node.selectNode();
    } else {
     System.out.println("Rule Not Linked to a Single Node");
    }
    break;
   case CONSTANT.ACTION_DING :
    if (this.audioClip != null && playing == false) {
     playing = true;
     this.audioClip.play();
     playing = false;
    }
   default: break; // do nothing
   }
  } else {
   switch(this.actionCode) {
   case CONSTANT.ACTION_NODEBOXED:
    this.node.setBoxed(false); break;
   case CONSTANT.ACTION_NODECOLOR:
//     this.node.unselectNode();
    this.node.setAlphaBox(false);
    break;
   }
  }


  this.setChanged();
  Thread th = new Thread() {
   public void run() {
    notifyObservers(sample);
    // Note: This is not asynchronous.
    //Will be a problem when more than one guys is listening
   }
  };
  th.start();
  // Asynchronously tell the others that "Hey - Display this guy"
 } // update ends

 private double processFilters(double newData) {
  double temp = newData;
  for (Filter f : filters) {
   temp = f.processNewData(temp); // stacks the filters
  }
  return temp;
 }

 protected void finalize() {
  if (this.sensor != null) {
   this.sensor.deleteSpecialObserver(this);
  }
```

```
package edu.media.resenv.clairvoyant;

import java.util.ArrayList;
import java.util.Observable;
import java.util.concurrent.ConcurrentHashMap;
public class CompoundRule extends SimpleRule implements SensorInterface{
 ArrayList<ComponentRuleDesc> ruleDescList;
 int operator = CONSTANT.OPERATOR_NOTVALID;
 ComponentRuleDesc firstComponentDesc; // First Component Rule
 ConcurrentHashMap <Observable, Boolean> seenSoFar
     = new ConcurrentHashMap<Observable, Boolean>();
 ConcurrentHashMap <Observable, ComponentRuleDesc> ruleMap
     = new ConcurrentHashMap<Observable, ComponentRuleDesc>();
 String name;
 public CompoundRule() {
  super(null);
  this.ruleDescList = new ArrayList<ComponentRuleDesc>();
  this.name = "Compound Rule Being Created";
  this.node = null; // Not linked directly to a node
 }
 int nonFirstRuleCount = 0;
 // Called for each component.
 // This should only be called before calling delayed check rule
 public void addComponentRule(ComponentRuleDesc e, int operator) {
  try {
   this.operator = operator;
   this.ruleDescList.add(e);
   this.ruleMap.put(e.rule, e); // Maybe they ran out of space.
   if (e.beginTimeOffset == 0 && e.endTimeOffset == 0) {
    System.out.println("Found the Parent Rule");
    this.firstComponentDesc = e; // This is the first component rule
   } else {
    this.nonFirstRuleCount++;
   }
   e.rule.addObserver(this);
  } catch (RuntimeException e1) {
   // TODO Auto-generated catch block
   System.out.println("Exception in addComponent Rule");
   e1.printStackTrace();
  }
 }
double lastOnTimeParent = 0;
/**
 * @param newSample
 * @param rule
 * @return boolean
 * Note that this function only checks if event with positive delays
 */
protected boolean delayedCheckRule(Sample newSample, Observable rule) {
 double newData = newSample.data;
 if (newData <1) {
  // its false.
  //Nothing could have possibly become true in the meantime
  return false; // There cant be a change
 }
 if (newData == 1.0) {
  if ((SimpleRule) rule == this.firstComponentDesc.rule) {
   // same rule, it is indeed the first guy
   this.lastOnTimeParent = newSample.timeStamp;
   for (Observable d: this.seenSoFar.keySet()) {
    this.seenSoFar.remove(d); // Not seen
   }
  }
  // See if it actually satisfies the time lag
  double beginTimeOffset = this.ruleMap.get(rule).beginTimeOffset;
  double endTimeOffset = this.ruleMap.get(rule).endTimeOffset;
  if ( (this.lastOnTimeParent + beginTimeOffset < newSample.timeStamp)
   && (this.lastOnTimeParent + endTimeOffset > newSample.timeStamp)) {
   this.seenSoFar.put(rule, true);
  }
  if (this.seenSoFar.size() == this.nonFirstRuleCount) {
   System.out.println("Returning True");
   return true;
  }
 }
 return false;
}
protected boolean checkRule(Sample newSample, Observable rule) {
 // will be changed by the other guy
  boolean resultOR = false;
  boolean resultAND = true;
  Sample lastSample;
  double timeZero = 0;
  double timeDelta = 0;

  if (this.ruleDescList.size() == 0) {
   System.out.println
   ("No rule set in complex rule, remember to call addComponentRule");
  }
  boolean resultXOR =
   ((this.ruleDescList.get(0)).rule.getLastSample().data==1.0)
             ? true:false;
  int count = 0;
  try {
   for (ComponentRuleDesc c: this.ruleDescList) {
    SimpleRule srule =  c.rule;
    lastSample= srule.getLastSample();
    boolean lastValue = (lastSample.data ==1.0) ? true:false;
    resultOR = resultOR || lastValue;
    resultAND = resultAND && lastValue;
    if (count > 0) {
     resultXOR = ((!resultXOR) && lastValue)
     || (resultXOR && (!lastValue)); // Not Tested
    }
    count++;

    if (timeZero == 0) {
     timeZero = lastSample.timeStamp;
    } else {
     timeDelta = lastSample.timeStamp - timeZero;
    }
   }
  } catch (Exception e) {
   System.out.println(e.getMessage());
  }
  if (this.operator == CONSTANT.OPERATOR_OR) {
   return resultOR;
  }
  if (this.operator == CONSTANT.OPERATOR_AND) {
   return resultAND;
  }
  if (this.operator == CONSTANT.OPERATOR_XOR) {
   return resultXOR;
  }
  return false;
 }
};
```

```
package edu.media.resenv.clairvoyant;

/**
 * Class ComponentRuleDesc, used as a component for rules
 * @author Manas Mittal
 *
 */

public class ComponentRuleDesc {

 SimpleRule rule; // you can also place a complex rule in it
 double beginTimeOffset;
 double endTimeOffset;

 ComponentRuleDesc(SimpleRule rule, double beginTimeOffset, double endTimeOffset) {
  this.rule = rule;
  this.beginTimeOffset = beginTimeOffset;
  this.endTimeOffset = endTimeOffset;
 }
}
```

```
package edu.media.resenv.clairvoyant;

public class smoothFilter extends Filter {

 double []data;
 int window;
 double currentSum = 0;
 int samplesSeen = 0;
 int latestIndex = -1;

 public smoothFilter(int window) {
  // TODO Auto-generated constructor stub
  this.window = window;
  data = new double[window];
  samplesSeen = 0;
 }

 public void changeParam(int newLength) {
  double[] newData = new double[newLength - 1];
  if (newLength > this.window) {
   System.arraycopy(this.data, 0, newData, 0, this.data.length);
   if (this.data.length == 0) {
    newData[0] = 0.0;
   }
   this.samplesSeen = data.length - 1;

  }
  else {
   System.arraycopy(
     this.data, this.data.length - newLength - 1,
     newData, 0, newLength);
  }
  this.data = newData;
 }

 @Override
 public double processNewData(double newData) {
  if (samplesSeen < data.length - 1) {
   samplesSeen++;
   latestIndex++;
   data[latestIndex % data.length] = newData;
   currentSum += newData;
   return CONSTANT.DBLNOTVALID;
  }
  else {
   latestIndex++;
   currentSum -= data[latestIndex%data.length];
   data[latestIndex] = newData;
   currentSum +=newData;
   return currentSum/data.length;
  }
 }

 public double getCurrentAvg() {
  return currentSum/data.length;
 }
}
```

Bibliography

[1] Wikipedia, the free encyclopedia. http://www.wikipedia.org. Accessed on 08/17/2008. 111

[2] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, January 2005. 35

[3] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003. 35

[4] Tarek Abdelzaher, Yaw Anokwa, Pter Boda, Jeff Burke, Deborah Estrin, Leonidas Guibas, Aman Kansal, Samuel Madden, and Jim Reich. Mobiscopes for human spaces. *IEEE Pervasive Computing*, 6(2):20–29, 2007. 29

[5] Adobe. Actionscript technology center. http://www.adobe.com/devnet/actionscript/. Accessed on 08/17/2008. 70, 80

[6] Adobe. Flash cs3 professional. http://www.adobe.com/products/flash/. Accessed on 08/17/2008. 70, 80

[7] The Zigbee Alliance. Zigbee. http://www.zigbee.org/en/index.asp. Accessed on 08/14/2008. 43, 75, 77

[8] Atmel. Avr32 32-bit microcontroller at32uc3b0256 datasheet. 76

[9] B. Bell, S. Feiner, and T. Hollerer. Information at a glance [augmented reality user interfaces]. *Computer Graphics and Applications, IEEE*, 22(4):6–9, Jul/Aug 2002. 30

[10] Ari Yosef Benbasat. An inertial measurement unit for user interfaces. Master's project, Massachusetts Institute of Technology, Department of Media Arts and Sciences, September 2000. 36

[11] Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47(1-3):139–159, January 1991. 64

[12] Declan Butler. Mashups mix data into global service. *Nature*, 439(7072):6–7, Jan 2006. 28

[13] W. Steven Conner, Lakshman Krishnamurthy, and Roy Want. Making everyday life easier using dense sensor networks. In *UbiComp '01: Proceedings of the 3rd international conference on Ubiquitous Computing*, pages 49–55, London, UK, 2001. Springer-Verlag. 29

[14] R. Graham Cooks, Zheng Ouyang, Zoltan Takats, and Justin M Wiseman. Detection technologies. ambient mass spectrometry. *Science*, 311(5767):1566–1570, Mar 2006. 24

[15] Allen Cypher, editor. *Watch What I Do: Programming by Demonstration*. MIT Press, 1993. 32

[16] Anind K. Dey, Raffay Hamid, Chris Beckmann, Ian Li, and Daniel Hsu. a cappella: programming by demonstration of context-aware applications. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 33–40, New York, NY, USA, 2004. ACM. 33

[17] S. Feiner, B. MacIntyre, T. Hollerer, and A. Webster. A touring machine: prototyping 3d mobile augmented reality systems for exploring the urban environment. In *Wearable Computers, 1997. Digest of Papers., First International Symposium on*, pages 74–81, 13-14 Oct. 1997. 30

[18] Steven Feiner, Blair Macintyre, and Dorée Seligmann. Knowledge-based augmented reality. *Commun. ACM*, 36(7):53–62, 1993. 30

[19] Quatre Sn E. Gamma, R. Helm, J. Vlissides, and I R Johnson. Design patterns: Elements of reusable object-oriented software. Book, 1995. 83

[20] Google. Google maps. http://www.maps.google.com. Accessed on 08/17/2008. 27

[21] Daniel Gyllstrom, Eugene Wu, Hee-Jin Chae, Yanlei Diao, Patrick Stahlberg, and Gordon Anderson. Sase: Complex event processing over streams, 2006. 35

[22] Thomas Haenselmann, Thomas King, Marcel Busse, Wolfgang Effelsberg, and Markus Fuchs. *Emerging Directions in Embedded and Ubiquitous Computing*, chapter Scriptable Sensor Network Based Home-Automation, pages 579–591. Springer Berlin / Heidelberg, 2007. 37, 46

[23] Björn Hartmann, Leith Abdulla, Manas Mittal, and Scott R. Klemmer. Authoring sensor-based interactions by demonstration with direct manipulation and pattern recognition. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 145–154, New York, NY, USA, 2007. ACM. 32, 33

[24] Charles R. Hildreth. *Intelligent Interfaces and Retrieval Methods for Subject Searching in Bibliographic Retrieval Systems.* Cataloging Distribution Service, Library of Congress, Washington, DC 20541., 1989. 61

[25] Harlan Hile and Gaetano Borriello. Information overlay for camera phones in indoor environments. In Jeffrey Hightower, Bernt Schiele, and Thomas Strang, editors, *Location- and Context-Awareness*, volume 4718 of *Lecture Notes in Computer Science*, pages 68–84. Springer, 2007. 31

[26] The Internet Movie Database (IMDb). Startrek, television series (196601969). http://www.imdb.com/title/tt0060028/. Accessed on 08/14/2008. 23

[27] Texas Instruments. System-on-chip for 2.4 ghz zigbee(tm) /ieee 802.15.4 with location engine (rev. b). http://focus.ti.com/lit/ds/symlink/cc2431.pdf. Accessed on 08/17/2008. 77

[28] Yuri Ivanov, Christopher Wren, Alexander Sorokin, and Ishwinder Kaur. Visualizing the history of living spaces. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1153–1160, 2007. 28

[29] Abstract Window Toolkit Java Programming Language. java.awt class robot. 57, 68

[30] Jeff Jetton. Tricorder (palm pilot freeware). http://www.jeffjetton.com/tricorder/index.html. Accessed on 08/17/2008. 24

[31] Ishwinder Kaur. Openspace: Enhancing social awareness at the workplace. Master's project, Massachusetts Institute of Technology, Department of Media Arts And Sciences, June 2007. 28, 78

[32] Sukun Kim, Shamim Pakzad, David Culler, James Demmel, Gregory Fenves, Steve Glaser, and Martin Turon. Wireless sensor networks for structural health monitoring. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 427–428, New York, NY, USA, 2006. ACM. 27

[33] Kurt Kleiner. The star trek tricorder. *New Scientist Blogs*, 2007. Accessed on 08/17/2008. 23

[34] Greg Kuchmek. Real tricorders. stim.com/Stim-x/0996September/Sparky/tricorder.html. Accessed on 08/17/2008. 24

[35] Mathew Laibowitz. Phd proposal: Distributed narrative extraction using imaging sensor networks, April 2007. 41, 42, 58, 65, 76, 77, 78, 79

[36] P. Levis, S. Madden, J. Polastre, R. Szewczy, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. Tinyos: An operating system for sensor networks. *Ambient Intelligence*, 2005. 35

[37] Philip Levis and David Culler. Mate : a tiny virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(5):85–95, December 2002. 69

[38] Henry Lieberman, editor. *Your Wish is My Command: Giving Users the Power to Instruct their Software*. Morgan Kaufmann, 2000. 32

[39] Joshua Lifton. *Dual Reality: An Emerging Medium*. Ph.D. Dissertation, Massachusetts Institute of Technology, Department of Media Arts and Sciences, September 2007. 19, 21, 59

[40] Joshua Lifton. *Dual Reality: An Emerging Medium*. Ph.D. Dissertation, Massachusetts Institute of Technology, Department of Media Arts and Sciences, September 2007. 24, 25

[41] Joshua Lifton, Mark Feldmeier, Yasuhiro Ono, Cameron Lewis, and Joseph A. Paradiso. A Platform for Ubiquitous Sensor Deployment in Occupational and Domestic Environments. In *Proceedings of the Sixth International Symposium on Information Processing in Sensor Networks (IPSN)*, pages 119–127, April 2007. 25

[42] Joshua Lifton, Mark Feldmeier, Yasuhiro Ono, Cameron Lewis, and Joseph A. Paradiso. A Platform for Ubiquitous Sensor Deployment in Occupational and Domestic Environments. In *Proceedings of the Sixth International Symposium on Information Processing in Sensor Networks (IPSN)*, pages 119–127, April 2007. 59

[43] Joshua Lifton, Manas Mittal, Michael Lapinksi, and Joseph A. Paradiso. Tricorder: A mobile sensor network browser. In *proceedings of the ACM CHI 2007 Conference - Mobile Spatial Interaction Workshop*, April 2007. 24, 26

[44] Alan L. Liu, Harlan Hile, Henry Kautz, Gaetano Borriello, Pat A. Brown, Mark Harniss, and Kurt Johnson. Indoor wayfinding:: developing a functional interface for individuals with cognitive impairments. In *Assets '06: Proceedings of the 8th international ACM SIGACCESS conference on Computers and accessibility*, pages 95–102, New York, NY, USA, 2006. ACM. 31

[45] Samuel Madden. *The Design and Evaluation of a Query Processing Architecture for Sensor Networks.* PhD thesis, University of California, Berkeley, 2003. 35, 69

[46] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, March 2005. 35

[47] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, New York, NY, USA, 2002. ACM. 27, 29

[48] Miklos Maroti, Gyula Simon, Akos Ledeczi, and Janos Sztipanovits. Shooter localization in urban terrain. *Computer*, 37(8):60–61, 2004. 29

[49] David Merrill. Flexigesture: An sensor-rich real-time adaptive gesture and affordance learning platform for electronic music control. Master's project, Massachusetts Institute of Technology, Department of Media Arts And Sciences, June 2004. 33, 34

[50] David Merrill and Joseph A. Paradiso. Personalization, expressivity, and learnability of an implicit mapping strategy for physical interfaces. In *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 145–154, New York, NY, USA, 2005. ACM Press. 33, 34

[51] Nelson Minar, Matthew Gray, Oliver Roup, Raffi Krikorian, and Pattie Maes. Hive: Distributed agents for networking things. In *Proceedings of ASA/MA'99, the First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents*, 1999. 37

[52] M.M. Molla and S.I. Ahamed. A survey of middleware for sensor network and challenges. In *Parallel Processing Workshops, 2006. ICPP 2006 Workshops. 2006 International Conference on*, page 6pp., 14-18 Aug. 2006. 69

[53] Rene Mueller, Gustavo Alonso, and Donald Kossmann. SwissQM: Next Generation Data Processing in Sensor Networks. In *Third Biennial Conference on Innovative Data Systems Research (CIDR 2007)*, Asilomar, CA, January 2007. 35

[54] Suman Nath, Jie Liu, and Feng Zhao. Sensormap for wide-area sensor webs. *Computer*, 40(7):90–93, 2007. 27, 28

[55] Inc. Nokia. Nokia 810: Internet tablet. http://europe.nokia.com/A4568593, 2008. Accessed on 08/10/2008. 81

[56] Philips Semiconductor. The I2C bus specification. http://www.nxp.com/acrobat_download/literature/9398/39340011.pdf, July 2008. Accessed on 07/14/2008. 76

[57] Till Quack, Herbert Bay, and Luc Van Gool. Object recognition for the internet of things. *The Internet of Things, Lecture Notes in Computer Science*, pages 230–246, 2008. 31

[58] Ted Selker. A bike helmet built for road hazards. http://news.cnet.com/2300-1008_3-6111157-1.html?hhTest=1, August 2006. Accessed on 08/12/2008. 33

[59] StreetLine. Streetline, city infrastructure technologies (streetlinenetworks.com). http://www.streetlinenetworks.com/site/index.php, July 2008. Accessed on 07/17/2008. 27

[60] G. Takacs, V. Chandrasekhar, N. Gelfand, Y. Xiong, W-C. Chen, T. Bismpigiannis, R. Grzeszczuk, K. Pulli, and B. Girod. Outdoors augmented reality on mobile phone using loxel-based visual feature organization. In *ACM International Conference on Multimedia Information Retrieval (MIR'08)*, 2008. 31

[61] Mark Weiser. The computer for the 21st century. *Scientific American*, pages 66–75, September 1991. 19

[62] Sean Michael White, Dominic Marino, and Steven Feiner. Designing a mobile user interface for automated species identification. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 145–154, New York, NY, USA, 2007. ACM. 75

[63] Wikipedia. Tricorder, From Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Tricorder, July 2008. Accessed on 07/14/2008. 23

[64] Jim Youll. Wherehoo and periscope: a time & place server and tangible browser for the real world. In *CHI '01: CHI '01 extended abstracts on Human factors in computing systems*, pages 109–110, New York, NY, USA, 2001. ACM. 31

[65] Degi Young and Ben Shneiderman. A graphical filter/flow representation of boolean queries: A prototype implementation and evaluation. *Journal of the American Society for Information Science*, 44:327–339, 1993. 61