Patch Panel: Enabling Control-Flow Interoperability in Ubicomp Environments

Rafael Ballagas
RWTH Aachen University
Department of Computer Science
Lehrstuhl für Informatik X
52056 Aachen, Germany
ballagas@cs.rwth-aachen.de

Andy Szybalski, Armando Fox
Stanford University
Department of Computer Science
353 Serra Mall, Stanford, CA 94305
andys, fox@cs.stanford.edu

Abstract

Ubiquitous computing environments accrete slowly over time rather than springing into existence all at once. Mechanisms are needed for incremental integration—the problem of how to incrementally add or modify behaviors in existing ubicomp environments. Examples include adding new input modalities and choreographing the behavior of existing independent applications. The iROS Event Heap, via its publish-subscribe coordination mechanism, provides the foundation for interoperation through event intermediation, but does not directly provide facilities for expressing these intermediations. The Patch Panel provides a general facility for retargeting event flow. Intermediations can be expressed as simple event translation mappings or as more complex finite-state machines. We describe an implemented prototype of the Patch Panel, including examples of its use drawn from real life applications in production use in the iRoom ubiquitous computing environment.

1. Introduction and Motivation

Weiser's landmark article defined ubiquitous computing environments as spaces where computers are both plentiful and subtle, allowing computation to blend invisibly into the fabric of our everyday activities [25]. Our homes [13] and office environments [10] will be augmented with technologies to improve the way we live and work. By examining the way these buildings evolve, it becomes clear that these ubiquitous computing environments will be incrementally deployed [20]. New technologies will be brought piecemeal into these spaces [4]. Clearly, system components like physical devices, applications, and network services cannot be expected to anticipate every other component they may encounter. These entities need to be able to coherently communicate without *a priori* knowledge of each other. Moreover, the nature of the interaction must be meaningful to both the individual components and the users of the system.

These constraints raise the questions: How does one design a system that may be augmented with future devices and services whose nature or feature set cannot be predicted in advance? Once the system is working, how does one integrate new devices and applications that may have no knowledge of each other's existence or function? We refer to this challenge as *incremental integration*, and we note that answering these questions would also answer the question of how to get *current* devices and services to interoperate that were not designed to do so.

2. Control-Flow Interoperability

The question of incremental integration has been central to our work in the iRoom, a conference-room-style ubicomp environment [10]. Our experience has led us to distinguish two main modes of interoperation: *data-flow* and *control-flow*. Data-flow interoperation refers to resolving mismatches in data type, a problem that received much attention in the mobile computing literature [7]. For example, a user may wish to connect a camera that produces images in JPEG format to a printer that accepts data in PostScript format. The iROS DataHeap [12] provides semi-transparent support for datatype transformation; Speakeasy [5] also provides such support by exploiting mobile code "proxies" to perform datatype conversion.

However, datatype conversion is insufficient to enable interoperation. Continuing the above example, the user may want to specify color vs. black-and-white

printing, choose paper size, or the order in which pages will print. The communication of metadata and commands between the camera and the printer will vary depending on the the specific devices and their capabilities. We refer to this challenge as *control-flow interoperability.*

To date, several methods have been proposed to address control-flow operability. One method is direct user intervention via GUI's as in iCrafter [19] and Speakeasy [5]: each time the user wishes to print a picture, a specialized GUI allows the user to set printing preferences for that specific printer model. This approach does not support making "automatic" connections free of user intervention, for example, using sensor data from a motion detector to turn on the lights. Another approach to control-flow interoperability is interface standardization, as attempted by Sun's Jini [24]: each *class* of service or device (e.g. printers) must adhere to a set of standard interfaces, allowing substitution of any service or device in that class. This turns out to be surprisingly difficult, even for such a seemingly simple class of devices as printers: after more than two years in committee, the Jini specification for printers is still (as of January 2004) labeled "pending ratification". One may infer that in fact the definition of a printer is a moving target since the feature set keeps changing. As new services or features are created, they cannot be exploited by applications until their integration into existing programmatic interfaces has passed the standards-approval process, potentially leading to unacceptable delay in adoption of the new service or feature.

The third approach to control flow interoperability, embodied by the Patch Panel, is intermediation. This approach utilizes a decoupled communication model, such as event publish/subscribe, for inter-component communication. Implicit in the model is the ability to intercept and rewrite these event streams. The iRoom's core software component, the Event Heap [9], was specifically designed to allow for such intermediation. However, it does not directly provide any facilities for expressing intermediation.

The Patch Panel fills this gap by providing intermediation facilities on top of the Event Heap. In essence, it provides a generic set of mechanisms for intercepting and translating incoming events to outgoing events, enabling control-flow interoperability among any set of entities that communicate via exchanging events over the Event Heap. Previous work has shown [9] that it is very easy to connect "legacy" desktop applications (such as Win32 applications that use OLE) and Web-based applications to the Event Heap, so that they can communicate using a common substrate; the Patch Panel leverages this ease of integration and adds the necessary machinery to perform the event translations needed to connect such components together. For the rest of this paper, we will generally focus on Event Heap-aware applications, although the Patch Panel approach is effective for any system using decoupled communication.

Systems like Context Toolkit [21] and Smart-its [3] combine information from multiple devices into abstracts of context cues. These context cues are essentially event aggregates that can be mapped to device or service control. Others have demonstrated the effectiveness of using state machines to express aggregate event composition for publish / subscribe systems [15, 17]. The similar capabilities included in the Patch Panel may also be used for this purpose, but this work sets out to show that they can be used for intermediation and incremental integration.

The rest of the paper proceeds as follows. We begin with a basic description of the intermediation facilities provided by the Patch Panel. Next we describe more formally how the notation used in the basic examples actually maps to the intermediation machinery available in the Event Heap. We then describe several programming patterns that arise in addressing control-flow interoperability and illustrate how the Patch Panel addresses each one; descriptions is are based on real-life examples from existing iRoom applications including the iClub [22] and the Workspace Navigator [8]. Finally, we discuss Patch Panel deployment scenarios, including the configuration interfaces available for both expert/administrator users and casual users.

3. The Patch Panel and Intermediation

3.1. Review of iROS and the Event Heap

For readers unfamiliar with the Event Heap, the central coordination mechanism used in the iROS ubicomp software framework, we briefly review its salient characteristics; details of its design can be found in [9], and an overview of how it enables inter-application coordination in the iRoom ubiquitous computing environment can be found in [10]. The Event Heap (EH) uses a tuplespace abstraction to provide, among other things, event publish/subscribe to a set of clients. An event consists of an unordered set of attribute-value pairs, one of which is designated as the event's *type* and the remainder of which are application-specific. The basic operations are `put`, which posts an event, and `get`, which queries for the existence of an event based on a template that specifies required fields and constraints on their values. Event subscription is also provided:

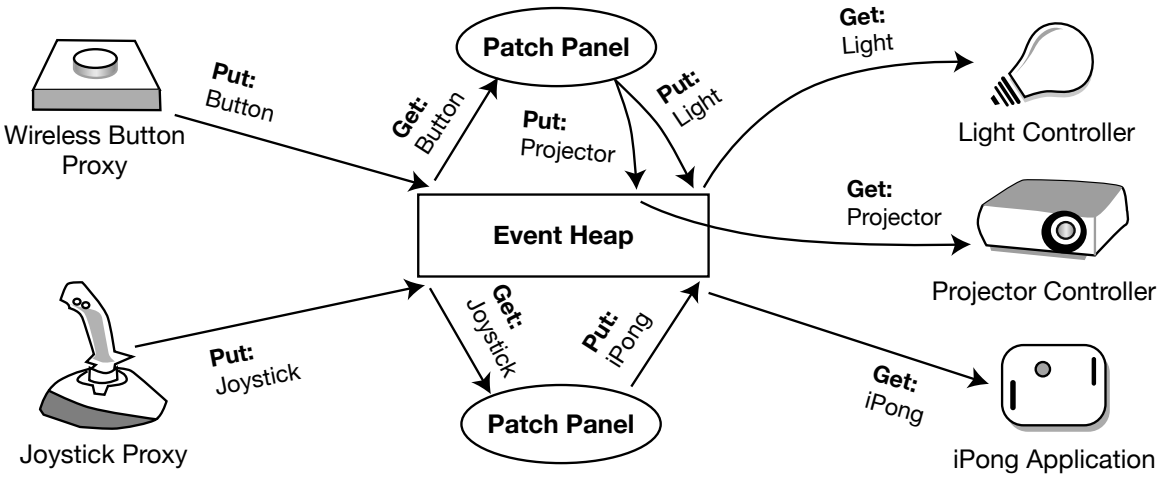

Figure 1. The Patch Panel adds a level of indirection to the communication channel to two components in order to perform event intermediation. The publish/subscribe semantics are also demonstrated. Note that there is actually only one Patch Panel process per Event Heap; two are shown above for visual clarity.

clients can receive a callback when events matching a particular template are posted to the EH. In keeping with the Boundary Principle [14], a single EH is the locus of interaction for a single ubiquitous computing environment, and a service or device can participate in that environment if and only if it can communicate with that environment's EH. Various libraries and other software components allow EH clients to be written in Java, C/C++, Visual Basic, Perl, Python, and other languages; servlets allow Web-based clients to perform limited EH operations as well.

3.2. Event Translations as Mappings

The Patch Panel enables intermediation among entities that communicate via event publish/subscribe (See Fig. 1). Simple intermediations are expressed as mappings that connect "triggers" (observed events emitted by some entity) with "outputs" (new events emitted as a result of the trigger, presumably for consumption by a different entity). The Patch Panel works by subscribing to all events. If an event is received that matches a trigger condition, the Patch Panel generates the corresponding output event(s) and posts them to the EH.

A *mapping* (trigger → output events) is the basic abstraction provided by the Patch Panel. For example, suppose we have a wireless iStuff button [1] that posts a *Button* event[1] each time it is pressed with string-valued field `id`, a light dimmer service that responds to *Lights* events containing an integer-valued field `brightness` between 0 and 10, and a projector-control service that responds to events *Projector* with boolean-valued field `powerOn`. We can configure the button to turn the lights and projector on by establishing the Patch Panel mapping in Figure 2.

Now, when someone presses the button, the following sequence of operations occurs:

1. In response to being pressed, the button posts a *Button*(`id=red`) event to the EH.[2]
2. Although neither the light service nor the projector service recognize the *Button* event, the Patch Panel recognizes it as a trigger for the *Button* mapping shown above.
3. The mapping fires, causing the Patch Panel to post the events *Lights*(`brightness=10`) and *Projector*(`powerOn=true`) to the EH.
4. The light controller recognizes the *Lights* event and turns the lights on; the projector controller recognizes the *Projector* event and turns the projector on.

3.3. Intermediation as State Machines

Suppose instead that we want the red button to *toggle* the lights and the projector between on and off. Assuming (as is often the case) that we cannot "query"

1 The stylized event names such as *Button* and *Lights* represent the mandatory event type field of the events.

2 As described in [1], simple hardware devices such as buttons and sliders typically communicate with a software *proxy* that connects the devices to the EH.

Button(`id=red`) → *Lights*(`brightness = 10`), *Projector*(`powerOn=true`)

Figure 2. Simple Patch Panel mapping that turns on lights and projector

the light service or the projector service to inquire about their current state (on or off), we could turn the button into a toggle switch by using a finite state machine (FSM) such as in Fig. 3.

If the red button is pressed while the Patch Panel is in the Off state, it will turn on the lights and the projector, and vice versa. Although the button is stateless and the light and projector services do not expose their internal state, the Patch Panel instantiates an FSM that "remembers" the current state of the interaction and responds accordingly to the *Button*(`id=red`) trigger event.

4. FSM's and Mappings

In this section we describe how the state machines used in the foregoing examples are actually translated to event mappings, and how the mappings are modified on each state transition. We also discuss the Patch Panel's performance and fault-tolerance.

4.1. Mappings as State Machines

As illustrated by Figure 3, the effect of expressing interactions as FSM's is that the mapping of a given trigger to a set of output event(s) depends on the current state of the FSM. In fact, such interactions are implemented using the same underlying mechanism that is used for simple mappings like the Button examples in section 3.2. To illustrate how this is done, consider a more complex example. Here we add support for an in-room motion sensor that posts a MotionSensor event whenever it detects activity. If no motion is detected for `timeout` seconds, the lights and projector turn off automatically. The FSM description is shown in Figure 4.

Using Figure 4, we can formalize the Patch Panel's operation as follows. Let $S_0, \ldots, S_{n-1}$ be the n states of an FSM that expresses a Patch Panel-mediated interaction. From Figure 4, let S_0 be the Off state and S_1 the On state. Let m_i be a simple event mapping of the form $t_i \rightarrow o_{i1}, o_{i2}, \ldots o_{ik}$, where t_i is a trigger event and $o_{i1} \ldots o_{ik}$ are the output events triggered by t_i. Finally, let $NS(i)$ be the number of the next state to go to after emitting the output events of mapping m_i. From Figure 3, each of the "on" clauses generates one mapping m_i and one next-state $NS(i)$ as demonstrated in Figure 5.

Let M_i be the set of mappings consistent with the FSM being in state S_i. For Figure 4, we would have $M_0 = \{m_0\}$ and $M_1 = \{m_1, m_2, m_3\}$. The Patch Panel maintains a single set P of mappings that is currently active, i.e. against which incoming events will be checked for triggers. When a state transition occurs into state S_i, P must be set to M_i. With this notation, we can express in pseudocode the operation of the Patch Panel (by convention, S_0 is the initial state):

1. Compute all m_i, $NS(i)$, and sets $M_0 \ldots M_{n-1}$ from textual FSM description;
2. Set P to M_0;
3. do forever:
   (a) wait for an event t_i that triggers some mapping m_i in P;
   (b) emit the mapping's output events $o_{i1}, \ldots, o_{ik}$;
   (c) set P to $M_{NS(i)}$;

In other words, the Patch Panel's textual front-end takes a description of a Mealy-style FSM and generates the corresponding per-event mappings to implement that FSM. As we will show, therefore, the Patch Panel can be configured either by modifying individual mappings using a GUI editor or by creating a textual FSM description such as those above. The FSM description language is loosely based on the intermediate-file format used by *fsmc*, a finite-state machine compiler used in several digital design courses. *fsmc* converts FSM specifications to sum-of-products Boolean expressions that can be used to program logic arrays. The only control-flow keywords we provide are `state` to declare a new state, `on` to specify trigger events, `send` to emit an output event, and `goto` to transition to the next state. A complete description of the language can be found in [2].

We now discuss the idiosyncrasies of running an FSM engine on top of the Event Heap.

4.2. PPMapping Events

The actual modification of the set P referred to above is itself controlled by events. In particular, the Patch Panel looks for a special event type called *PPMapping*. These events can be used to add, remove and modify the active mappings P.

```
state Off {
    on Button(id = red) {
        Lights(brightness = 10);
        Projector(powerOn = true);
        goto On;
    }
}
state On {
    on Button(id = red) {
        Lights(brightness = 0);
        Projector(powerOn = false);
        goto Off;
    }
}
```

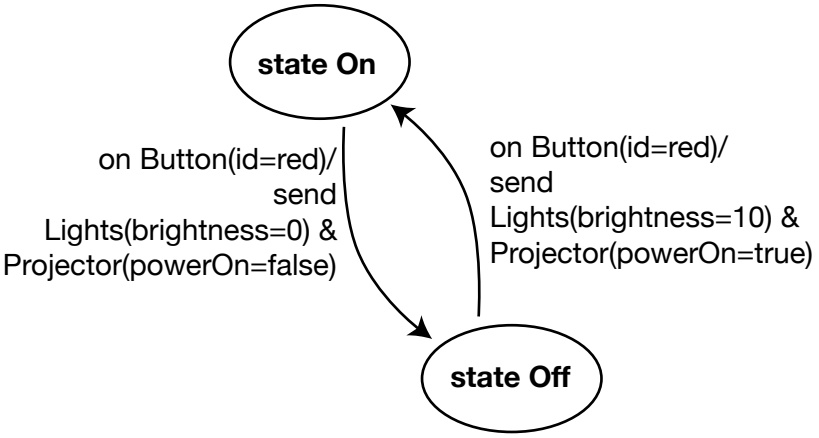

Figure 3. Simple light-and-projector toggle

```
state Off {
    on Button(id = red) {             // turn things on manually
        Lights(brightness = 10);
        Projector(powerOn = true);
        goto On;
    }
}
state On {
    on MotionSensor {                 // motion detected, so...
        set timer 1000*timeout;       //   ...reset idle timer
        goto On;
    }
    on timer {                        // no motion sensed, so...
        Lights(brightness = 0);       //    ...turn things off
        Projector(powerOn = false);
        goto Off;
    }
    on Button(id = red) {             // turn things off manually
        Lights(brightness = 0);
        Projector(powerOn = false);
        goto Off;
    }
}
```

Figure 4. FSM description with timers

m_0	=	Button(`id=red`) → Lights(`brightness=10`), Projector(`powerOn=true`)	$\mathrm{NS}(0) = S_1$
m_1	=	MotionSensor → set timer 1000*timeout	$\mathrm{NS}(1) = S_1$
m_2	=	timer → Lights(`brightness=0`), Projector(`powerOn=false`)	$\mathrm{NS}(2) = S_0$
m_3	=	Button(`id=red`) → Lights(`brightness=0`), Projector(`powerOn=false`)	$\mathrm{NS}(3) = S_0$

Figure 5. Mappings that implement the state machine for in-room motion sensor

The reason for exposing PPMapping as an event-based interface (i.e. as an external primitive) is to allow dynamic reconfiguration: it allows the set of active mappings to be changed not only as a result of internal state transitions, but also by an external controller program. In section 5 we illustrate an example of how to exploit this capability in the user interface for programming and using the Patch Panel.

4.3. Atomic State Transitions

By design, the Event Heap does not guarantee ordering of events from different sources. Therefore, output events from the PatchPanel and trigger events from other sources may be interleaved. This is problematic since the FSM abstraction requires that the emitting of output events (step 3b above) and transition to the next state (step 3c) must occur atomically. To address this, the Patch Panel provides *event chains*, which are groupings of events that must be processed atomically. In our current implementation, any outgoing event containing a field `ChainEvent` is considered to be part of an atomic event chain. The Patch Panel's event emitter passes event chains directly to its own event handling loop (i.e. these events do not travel through the Event Heap at all), where they are given highest priority.

4.4. Performance and Failure Semantics

The Patch Panel is often in the critical path of the human interface action-perception loop. The basic measure of Patch Panel performance is the delay between the posting of a trigger event to the Event Heap and the receipt of the correct resulting output event(s) to the Event Heap. Our microbenchmark for simple mappings in Figure 6 shows this delay to be about 12ms, of which 6ms is due to the Patch Panel itself and the remainder is attributed to serialization/deserialization of events, network delay, and internal Event Heap delay. The total delay increases with events that contain more fields. It may also increase when there are many (over 100) mappings, because the Patch Panel's mapping hash table was designed for small numbers of mappings.

Because the Patch Panel is an Event Heap client, it inherits the Event Heap's robustness [18] in handling failures. In particular, the failure of other Event Heap clients does not compromise the operation of the Patch Panel. Failure of the Patch Panel causes intermediation to cease until it is restarted. The Patch Panel's internal mappings are saved to disk, and the states of the active state machines will survive restarts of both the Event Heap and the Patch Panel.

5. Functionality Motivated by Example

One of the lessons of our work relates to the repertoire of "programming patterns" needed to cover a broad variety of incremental-integration scenarios. In this section, we describe some of the patterns supported by the Patch Panel, motivating each one with a real-life example. As a preview, the mechanisms we will describe are as follows:

1. Allowing for *dependent translations*, where the field values of output events are not known at the time the mapping is specified. Instead, the output fields must be derived from field values of trigger events at run time.
2. Programmatically exposing to applications the ability to change mappings on the fly, allowing the construction of GUIs and other applications whose function is to configure the Patch Panel itself.
3. Allowing the use of global variables (whose value persists across individual firings) to further support interactions requiring persistent state.
4. Providing an abstraction for time, allowing for time-based interactions

This set of examples is based on two real applications: the iClub [22], an iRoom application that creates an interactive dance club environment using the large displays in the iRoom, and Workspace Navigator (WSN) [8], an application that captures meetings in the iRoom using video cameras, digital whiteboard systems, screen capture software and a document archive. Videos showing each application in use are at `http://iwork.stanford.edu`.

iClub is a distributed application that includes a playlist program to select songs and sound effects, an audio proxy application that plays the audio and publishes a "beat clock" event synchronized to each beat of the music, several visualization applications that synchronize to the beat of the music by subscribing to the beat clock events, and a GUI that controls various aspects of the music as it plays

5.1. On-the-Fly Integration

The audio proxy recognizes events to adjust the volume and tempo of the music, inject sound effects, and apply high- and low-pass frequency filters (a common audio special-effect) to the music. In the original version of iClub, a human DJ would use a GUI to control

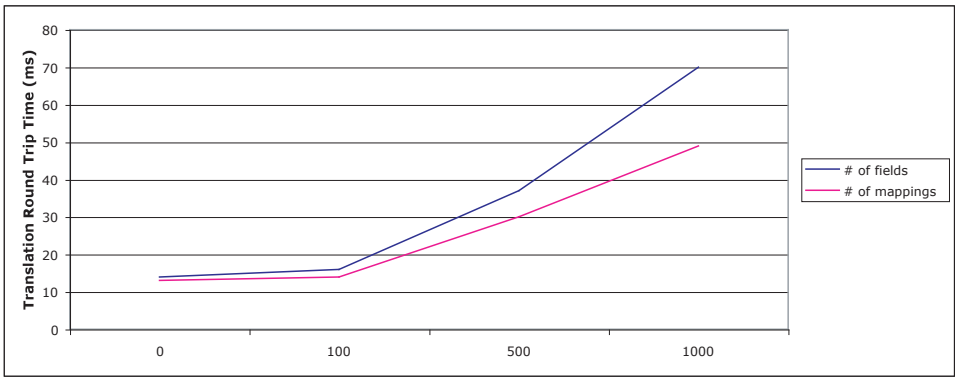

Figure 6. Patch Panel performance degradation. The performance degrades as the number of oustanding mappings increases, and as the number of event fields increases.

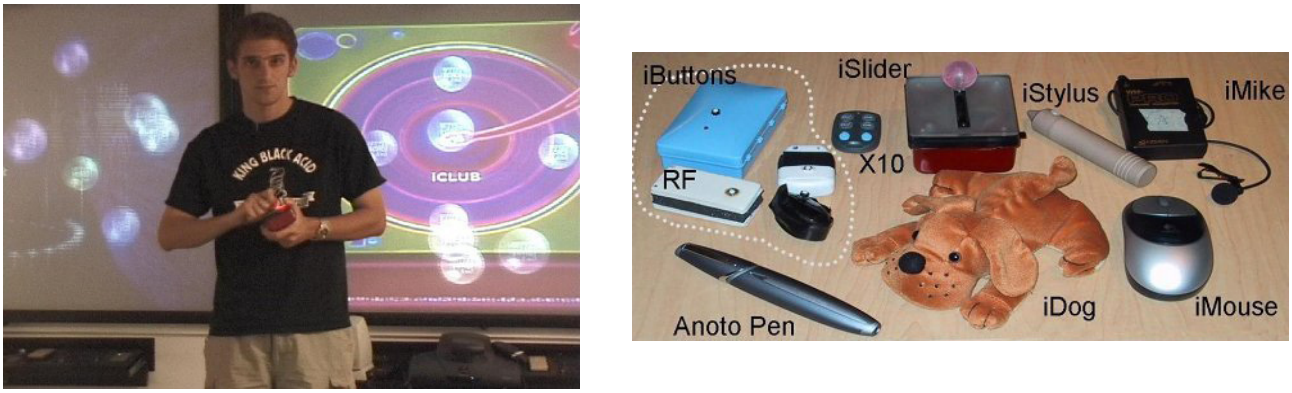

Figure 7. (Left)the iClub in Action with iSlider, (Right)examples of iStuff [1] input devices

these aspects of the music. However, an observer suggested that the clubgoers themselves should be able to participate in the music creation without leaving the dance floor. We arranged to give each clubgoer a wireless button with a unique ID as they entered the room. Each button was mapped through the Patch Panel to direct the audio proxy to play a sound, so each clubgoer had her own characteristic sound effect that could be injected by pressing her button. We refer to this common programming pattern—connecting a new physical or other UI to an existing behavior—as *on-the-fly integration.*

5.2. Range Normalization

The developers' next inspiration was to allow a DJ to use a wireless handheld slider to have mobile control the tempo of the current song. Conceptually, this is similar to the previous example in that a slider event must be used to trigger an iClubAudio output event, except that the *value* of the slider must also be reflected in the output event. Furthermore, the specific slider we used produces real numbers in the range 0.0 to 1.0, whereas the iClubAudio's `tempo` parameter must be an integer from -10 to $+10$. We use the term *range normalization* to describe this pattern for incremental integration. The Patch Panel supports range normalization by allowing output event fields to reference input event fields and by providing a simple arithmetic expression evaluator. Figure 8 shows a mapping that connects the slider to the tempo control.

This mapping will fire when the Patch Panel receives an event of type *iSlider* that contains a field named `position`. The `tempo` field of the *iClubAudio* output event is computed from the `position` field of the input event (`in.`) each time the mapping fires. Similarly, the prefix `out.` could be used to reference other fields in the current output event; the Patch Panel automat-

iSlider(`position = *`) → iClubAudio(`tempo = (int)in.position * 20 - 10`)

Figure 8. Range Normalization and Equation Specification

ically detects syntax errors or output field dependency loops and leaves such equations unresolved (in string form) for debugging purposes.

5.3. Dynamic Reconfiguration

The wireless slider works well to control a single parameter, but in order for a DJ or clubgoer to abandon the desktop GUI completely, she must have the ability to change *any* of the music parameters, not just the tempo. We constructed a new device by physically attaching four wireless buttons (call them 1, 2, 3, 4) to the slider. Pressing a button determines which music parameter—tempo, volume, high pass filter, low pass filter—is controlled by the slider. This case is more subtle, because the effect of pressing a button is not to emit a new audio event, but rather to affect the handling of future iSlider events. In other words, pressing a button should change the currently-active iSlider mapping. We therefore refer to this pattern as *dynamic reconfiguration*.

As explained previously, the Patch Panel consumes events of type *PPMapping* to modify the set of currently active mappings. With this in mind, the mappings to implement the "multi-slider" handheld music controller are shown in Figure 9.

When the Patch Panel receives a *Button*(`id=1`) event, for example, it emits a *PPMapping* event that will set up a new mapping for future *iSlider* events to control the music's tempo. This powerful dynamic reconfiguration allows complex interactions to be synthesized without direct user interaction.

Since the above mappings are somewhat difficult to interpret for a programmer unfamiliar with the PatchPanel, we could also have expressed the multi-slider example using the FSM representation shown in Figure 10, and in fact, submitting this FSM description to the Patch Panel's FSM compiler would effectively result in the mappings in Figure 9.

Several points about this example should be emphasized. First, the handheld DJ device and the interaction with the iClub Audio Proxy were assembled from devices and services that had no *a priori* knowledge of each other. In other words, the slider has no concept of a button or vice versa, and the iClub Audio Proxy has no concept of any physical devices. Also in this example, the buttons and mappings, once initially set up by an administrator, represent a tangible UI for reconfiguring the Patch Panel that can be used by people who have no technical skill or knowledge of Event Heap operation. We return to the issue of how different users might actually use the Patch Panel in section 6.

5.4. Semantic Mismatches and Globals

At one point, the wireless slider malfunctioned, and although we did not have another slider immediately available, we did have a joystick. However, while a slider's position naturally maps to the value being controlled, a joystick's position naturally maps to the rate of change of the value being controlled (since joysticks are self-zeroing). We refer to this circumstance as *semantic mismatch*, and in this case can be resolved by using an FSM with global variables as shown in Figure 11. Global variables store values that must persist across firings of mappings; variables can be set using "PPVariable" events and dereferenced on the output side of any mapping by using the prefix `global`. To mimic the slider's behavior with a joystick, we use a global variable to hold the current "slider position" and adjust the global variable's value each time the joystick is moved. To avoid wild fluctuation in the variable's value, we can use timers (as previously described) to control the interval at which the joystick is sampled; changing the timer value changes the sensitivity of the joystick as a controller. Although the joystick is an imperfect interaction modality for the iClub Audio Proxy, the Patch Panel made it possible to use it as an adequate substitute until the slider could be replaced.

5.5. Integration in Workspace Navigator

The last example concerns Workspace Navigator (WSN) [8], an application that captures multi-person meetings in interactive rooms containing shared public displays. WSN's GUI provides a "bookmark" feature that allows a participant to flag an important moment in the meeting; WSN's meeting-replay tools can then be used later to reconstruct the state of the meeting (e.g. which documents were visible on each of the shared displays) at the time the bookmark was inserted. To implement the bookmark function, the WSN GUI console sends a *Bookmark* event to the WSN server application when the Bookmark GUI widget is clicked.

Button(id=1) → (PPMapping[iSlider(position = *) → iClubAudio(tempo=(int)in.position*20-10)])
Button(id=2) → (PPMapping[iSlider(position = *) → iClubAudio(volume=(int)in.position*100)])
Button(id=3) → (PPMapping[iSlider(position = *) → iClubAudio(highfreq=(int)in.position*100)])
Button(id=4) → (PPMapping[iSlider(position = *) → iClubAudio(lowfreq=(int)in.position*100)])

Figure 9. Patch Panel mappings that enable the multi-slider handheld music controller

```
state ControllingTempo {
  on iSlider(position=*) { send iClubAudio(tempo=int(in.position)*20-10); }
  on Button(id=2) { goto ControllingVolume; }
  on Button(id=3) { goto ControllingHighFreq; }
  on Button(id=4) { goto ControllingLowFreq; }
}
state ControllingVolume {
  on iSlider(position=*) { send iClubAudio(volume=int(in.position)*20-10); }
  on Button(id=1) { goto ControllingTempo; }
  on Button(id=3) { goto ControllingHighFreq; }
  on Button(id=4) { goto ControllingLowFreq; }
  }
}
...
```

Figure 10. FSM description of iClub multi-slider mappings

During user testing of WSN, one user complained that inserting a bookmark required disrupting the meeting to acquire the shared keyboard and mouse (in order to interact with the GUI), discouraging users from taking advantage of this feature. The researcher proposed giving each meeting participant a wireless button that could be discreetly pressed to add a bookmark during the meeting. This approach has the additional benefit that each bookmark could be associated with the participant who inserted it.

To implement this, an iRoom administrator created a simple Web-based "This Is My Button" wizard that configures an iButton (an iStuff wireless physical button) to send Bookmark events. On entry to the iRoom, a meeting participant, say Rachel, picks up an iButton from a bucket of buttons, enters her name into the Web form, and submits the form. The form submission runs a servlet that waits for the next button press from any iButton. Rachel now presses the (physical) button, causing the servlet to establish a Patch Panel mapping connecting her particular button to bookmark events with her name attached to them.

This is another example of dynamic reconfiguration. The entire process of integrating the wireless button interaction into the WSN project only took about an hour and did not require any changes to the Workspace navigator code or the wireless buttons.

5.6. Summary

It has been about a year since work began on the Patch Panel, and in that time, researchers in our space have been using it to put together separate components in ad-hoc ways. Uses have ranged from enhancing existing applications, as illustrated by the foregoing examples, to creating new interactions using physical input devices. All the examples above are real and are in production use. Although each is simple by itself, together they illustrate the versatility and relative simplicity of the Patch Panel approach to interoperation, allowing multi-device systems to be deployed or existing applications to be augmented in a matter of minutes. The table in Figure 12 summarizes the examples, including the interoperability programming pattern illustrated by each and the Patch Panel mechanism(s) leveraged to achieve the desired result.

6. Configuration and Ease of Use

The ease of configuring mappings for interoperation is a key consideration in evaluating the usefulness of the Patch Panel. Since Patch Panel mappings can be changed by any client that emits *PPMapping* events,

```
state JoystickMoved {
   on Joystick(joystickX, joystickY) {
       global.currentX += in.joystickX * global.scaleFactor;
       global.currentY += in.joystickY * global.scaleFactor;
       send Position(global.currentX, global.currentY);
       set timer sampleRate;
       goto WaitingForSample;
   }
}
state WaitingForSample {
   on timer { goto JoystickMoved; }
}
```

Figure 11. Resolving semantic mismatch between relative-position and absolute-position devices. We have minimally stylized the code for readability.

(Section) Example Name	On-the-fly Integration	Dynamic Reconfiguration	State Machine Representation	Equations / Range Normalization	Global Variables / Semantic Mismatch	Timers
(3.2) Lights and Projector on	x					
(4.1) Lights and Projector toggle	x	x	x			
(4.1) Motion sensor toggle	x	x	x		x	x
(5.1) iClub sound buttons	x					
(5.2) iClub slider	x			x		
(5.3) iClub multi-slider	x	x		x		
(5.4) iClub joystick	x	x	x	x	x	x
(5.5) Workspace Navigator	x	x				

Figure 12. Patch Panel Example Summary Table

we can create a variety of configuration interfaces, from narrowly-specialized GUI's for casual users to sophisticated GUI's or textual interfaces for administrators and expert users.

6.1. Support for Expert Users

We distinguish expert from casual users. Expert users include: developers who want to create or augment applications by connecting new behaviors via the Patch Panel; system administrators; and "power users" (akin to those who can write complex macros in spreadsheet programs) who want to modify the behavior of existing applications.

The FSM scripting language described in section 4.1 provides a powerful textual front-end for expert users. We have also built the Patch Panel Manager (figure 13), a tree-based view of the Patch Panel mappings that allows expert users to graphically browse the current mappings and create new ones. Using this GUI requires some understanding of Event Heap semantics as well as the event interfaces of individual components. In everyday situations, the Patch Panel

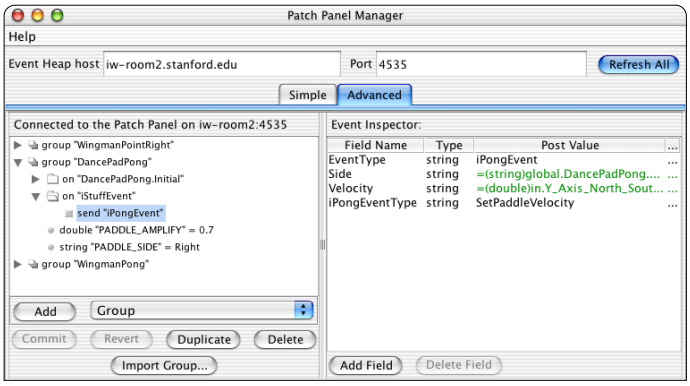

Figure 13. The Patch Panel Manager

Figure 14. Button-to-URL Configuration Servlet

Manager is used to visualize and debug the state of the Patch Panel. It is also particularly useful for making on-the-fly modifications to mappings or for "debugging" in-progress mappings; for example, changing the amplification factor applied to an input device.

6.2. GUI's for Casual Users

While the Patch Panel Manager GUI and the FSM scripting language satisfy most of an administrator's Patch Panel needs, it is also important for casual users of different skill levels to be able to define interactions to meet their needs. To demonstrate the potential simplicity of configuring the Patch Panel we have created a Button-to-URL Configuration Servlet (Fig. 14). It provides a web-based "wizard" interface that configures an iStuff button to display a user-chosen Web page. To configure a button, the user types the target URL into the configuration Web page and clicks "Submit". Then the servlet probes the Event Heap for the next button press. The user picks up any button from a bucket of buttons, and presses it. The servlet associates the pressed button with the URL; any subsequent events from that button will be translated to an event that opens the URL on a large public display.

This configuration servlet and the iClub multi-slider illustrate a class of interfaces that require developer effort up front to expose specific and limited Patch Panel functionality to less technical users. We are in the process of exploring more general "wizard" type configuration interfaces for casual users of iStuff [1].

7. Conclusions

We have argued from the beginning [18] that incremental integration is an essential property of robustly-evolving ubicomp environments, and that a powerful technique to enable integration is by intermediation of control flow. This insight was originally expressed in [15], but to the extent that intermediation is a solution in search of a problem, we believe that interoperation and evolution in ubiquitous computing environments is that problem.

Although an expert user can certainly write a full-blown Java program with arbitrarily complex event translation logic, we hypothesized that a small number of programming patterns serve to capture a wide variety of common integration and evolution tasks. Our goal was to provide a simple programming substrate that supports rapid and easy expression of the most common such patterns through its textual interface, and admits of the rapid creation of graphical or physical interfaces for novice users to perform

narrowly-specialized configuration tasks, as the iSlider and Workspace Navigator examples illustrate.

The result of our work to date is a system that enables intermediation-based control-flow interoperability in ubiquitous computing environments. Semantic mismatches between incompatible interfaces can be resolved with state machine and equation evaluation capabilities. The dynamic reconfiguration feature of the Patch Panel is critical in enabling state machine capabilities, retargeting event flow based on user input or context from sensor data, and enabling simple user interfaces to reprogram the Patch Panel. This combination of features provides a powerful integration tool that can be used to easily create and modify, often in a matter of minutes, interactions between networked hardware and software components in ubiquitous computing environments.

8. Acknowledgments

The authors thank Maureen Stone, Brad Johanson, Jan Borchers, and Shankar Ponnekanti for insightful comments and suggestions for this work. This research is supported by a National Science Foundation Graduate Research Fellowship and by a grant from the Wallenberg Global Learning Network. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Ballagas, R., Ringel, M., Stone, M., Borchers, J: iStuff: a Physical User Interface Toolkit for Ubiquitous Computing Environments. Proc. of CHI (2003)

[2] Ballagas, R., Szybalski, A., Fox, A.: "Overview of the Patch Panel Finite State Machine Description Language." Stanford Computer Science Technical Report (2003)

[3] Beigl, M., Gellersen, H. "Smart-Its: An Embedded Platform for Smart Objects", *Smart Objects Conference* (2003).

[4] Edwards, K., Grinter, R.: At Home with Ubiquitous Computing: Seven Challenges. Proc. of UBICOMP (2001) 256-272.

[5] Edwards K., Newman, M., Sedivy, J.: The Case for Recombinant Computing. Xerox Palo Alto Research Center Technical Report CSL-01-1 (2001)

[6] Edwards K. et al: Challenge: Recombinant Computing and the Speakeasy Approach. Proc. of MOBICOM (2002)

[7] Fox, A., Gribble, S.D., Chawathe, Y., Brewer, E.A.: "Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspectives". *IEEE Personal Comm.* (1998)

[8] Ionescu, A., Stone, M., and Winograd, T. "WorkspaceNavigator: Capture, Recall, and Reuse using Spatial Cues in an Interactive Workspace." Stanford Computer Science Technical Report 2002-04 (2002).

[9] Johanson, B. and Fox, A. The Event Heap: A Coordination Infrastructure for Interactive Workspaces. Proceedings of the 4th IEEE Workshop on Mobile Computer Systems and Applications (WMCSA-2002).

[10] Johanson, B., Fox, A., Winograd, T.: The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms. IEEE Pervasive Computing Magazine, Vol. 1, Issue 2, (2002) 71-78.

[11] Johanson, B., Hutchins, G., Stone, M., and Winograd, T. PointRight: Experience with Flexible Input Redirection in Interactive Workspaces. Proc. UIST (2002).

[12] Kícíman, E., Fox, A., Using Dynamic Mediation to Integrate CTOS Entities in a Ubiquitous Computing Environment. Second Annual Symposium on Handheld and Ubiquitous Computing (2000)

[13] Kidd, C. D., Orr, R. J., Abowd, G. D., Atkeson, C. G., Essa, I. A., MaclIntyre, B., Mynatt, E., Starner, T. E., Newstetter, W.: The Aware Home: A Living Laboratory for Ubiquitous Computing Research. Proc. of CoBuild. (1999)

[14] Kindberg, T., Fox, A., System Software for Ubiquitous Computing. IEEE Pervasive Computing 1(1).

[15] Munson, M: System Support for Composing Distributed Applications Using Events. Dissertation. Pembroke College, University of Cambridge (1998)

[16] Myers, B. Kosbie, D.: Reusable Hierarchical Command Objects. CHI 1996.

[17] Pietzuch, P., Shand, B., Bacon, J. A Framework for Event Composition in Distributed Systems. In Proc. Middleware (2003) 62-82.

[18] Ponnekanti, S., Johanson, B., Kiciman, E., and Fox, A. Portability, Extensibility and Robustness in iROS. In Proc. IEEE International Conference on Pervasive Computing and Communications (PerCom 2003), Dallas, TX, March 2003.

[19] Ponnekanti, S., Lee, B., Fox, A., Hanrahan, P., Winograd, T.: iCrafter: A Service Framework for Ubiquitous Computing Environments. Proc. of UBICOMP (2001)

[20] Rodden, T., Bedford, S.: The Evolution of Buildings and Implications for the Design of Ubiquitous Computing Enviroments. Proc. of CHI. (2003) 9-16

[21] Salber, D.,Dey, A., Abowd, G. "The Context Toolkit: Aiding the Development of Enabled Applications." *Proc. of CHI* (1999) 434-441.

[22] Samberg, J., Fox, A., Stone, M. "iClub, An Interactive Dance Club", *Ubicomp 2002 Video Program* (2002).

[23] Taylor, R., et al. A Component- and Message-Based Architectural Style for GUI Software. IEEE Transactions on Software Engineering, (1996)

[24] Waldo, J.: Alive and Well: Jini Technology Today. Computer Vol. 33, Issue 6. (2000) 107-109

[25] Weiser, M.: The Computer for the 21st Century. Scientific American, Vol. 265, Issue 3. (1991) 91-104.