Theoretical and Architectural Support for Input Device Adaptation

Jingtao Wang, Jennifer Mankoff
Computer Science Division
387 Soda Hall, UC Berkeley
Berkeley, CA 94720-1776, USA

{jingtaow, jmankoff}@cs.berkeley.edu

ABSTRACT

The graphical user interface (GUI) is today's *de facto* standard for desktop computing. GUIs are designed and optimized for use with a mouse and keyboard. However, modern trends make this reliance on a mouse and keyboard problematic for two reasons. First, people with disabilities may have trouble operating those devices. Second, with the popularization of wireless communication and mobile devices such as personal data assistants, the mouse and keyboard are often replaced by other input devices. Our solution is a tool that can be used to translate a user's input to a form recognizable by any Windows-based application. We argue that a formal model of input is necessary to support arbitrary translations of this sort. We present a model, based on Markov information sources, that extends past work in its ability to handle software-based input such as speech recognition, and to measure relative device bandwidth. We also present our translation tool, which is based on our model, along with four applications built using that tool.

Categories and Subject Descriptors

H.5.2 [**Information Interfaces and Presentation**]: User Interfaces – Theory and methods, Graphical User Interfaces H.1.2 [**Information Systems**]: User/Machine Systems – Human factors;

General Terms

Design, Human Factors, Standardization, Theory.

Keywords

User Interface, Input, Information Entropy, Input Adaptation, Input Transformation.

1. INTRODUCTION

The graphical user interface (GUI) is today's *de facto* standard for

desktop computing. GUIs are designed and optimized for use with a mouse and keyboard. However, modern trends make this reliance on a mouse and keyboard problematic for two reasons. First, people with disabilities may have trouble operating those devices. Second, with the popularization of wireless communication and mobile devices such as personal data assistants and cell phones, the mouse and keyboard are often replaced by other input devices.

This leads to a dilemma. UI widgets assume that a mouse and keyboard can be optimized to achieve better "direct manipulation," saving the GUI designer time and effort. However, a user with physical or visual impairments may need to access existing applications without the use of a keyboard or mouse.

It has been suggested in the past that specialized interfaces built to accommodate these needs are the best solution to this problem, and indeed in the realm of small mobile devices, this is the approach that has been taken. However, from a disability access perspective, this solution is not tenable. It leads to a situation in which the "accessible" version of a piece of software lags one or more versions behind the "standard" version, leaving those with disabilities at a further disadvantage.

Thus, our solution to this problem is to focus on tools and techniques that will save the time of designers or programmers for adapting an application to a more diverse set of input devices, including a range of physical devices and software such as speech recognition, without changing or accessing the application's source code.

As a first step towards solving this ambitious problem, we have developed a theoretical model to describe the bandwidth capacity of input devices and to estimate the bandwidth changes in input adaptation (*i.e.* using one or multiple devices to simulate another input device). We have also developed an application independent toolkit that can help developers of specialized input devices to connect them to legacy applications. The architecture embodied by our tool (the Input Adaptation Toolkit, or IAT) establishes an abstract layer between input devices and applications. This additional layer can be configured to transform events to a uniform internal state space format, and generate application level events to simulate the destination input devices that the application was originally designed for.

Although IAT does not make the adaptation process completely automatic and foolproof, it can save designers and programmers a

significant amount of time in making input device specific applications accessible to other input devices.

In the following section, we establish a theoretical foundation for input device adaptation by using information and coding theory [1]. In Section 3 we introduce the architecture of the Input Adaptation Toolkit (IAT), and describe the adaptation operations it provides, such as duplication, composition, and splitting. Section 4 presents four Input Adaptation applications (*Duo Keypad*, *T9 input method*, *virtual keyboard* and *Grid mouse*) based on IAT to demonstrate related features of IAT. We end with related work, followed by conclusions and future work.

2. MATHEMATICAL MODELING

In this section, we establish a theoretical model of input devices based on information entropy and information coding theory [1]. Based on this model, we derive a theorem, which supports quantitative analysis of input device adaptation.

From a high level, an input device can be thought of as a communication channel between a user and an application. The user expresses his/her intention to the computer via a series of operations, and the application responds to the user according to the input events received. An alternate interpretation is that the input process is a "flow" of information bits originating from the user and transferred via the input device to the application.

Not all input devices have the same capacity for transmitting information to the application. For example, the four cursor keys on a keyboard are less effective than a mouse for pointing tasks, and a person can enter text faster using a full size QWERTY keyboard than using the keypad in a cell phone. Researchers usually use the term "bandwidth" to describe the capacity of an input device [6][10].

While these comparisons are believable, they are inherently qualitative. Our contribution is a quantitative model of bandwidth that can be used to compare input devices. For example, a 10 key numeric keypad can be mathematically defined, using the equations presented next, to have a relative bandwidth of 3.32 bits per keystroke.

Note that this does not take into account the human bandwidth limitations relating to the use of fingers, hands and arms, but is solely a measure of the amount of information carried in each keystroke. We use the term *relative bandwidth* because our work currently assumes that the physiological difficulty of, say, pressing a cell phone key and pressing a keyboard key is equal. Past work has considered these physiological differences, and other factors such as whether a device is rotated or pressed and relative *vs.* absolute positioning [6]. However, that work could not easily model non-mechanical input channels such as speech recognition or an on-screen soft keyboard. Our model can handle recognition and other software-based input, and additionally can account for the selective frequency of use for different keys, words or other input events, something not handled by past models.

2.1 Modeling the input event generation process

From the information theoretic point of view, an input device can be seen as an information source emitting discrete symbols (input events) for an application (Figure 1). For example, the symbols emitted from a 10-key numeric keypad are – '1', '2', '3', ..., '9', '0', and the possible symbols from a discrete word speech recognizer with a vocabulary of 20 words are the 20 labels corresponding to those words. In addition to its symbol set, an input device is defined by the probability of each symbol occurring. The probability of a symbol occurring at any given moment in time is determined both by physical constraints (such as the inability of a mouse to jump across the screen), and usage constraints (such as the fact that the space bar in a QWERTY keyboard has a higher frequency in text transcription than most of the other keys). When each symbol is equally likely to occur, the probability for each would be $1/n$ where n is the number of symbols in S.
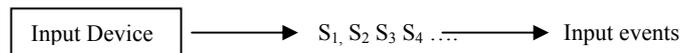

Figure 1. An input device represented as an information source

Restated in mathematical terms, suppose that an input device generates an event S_i with probability $P(S_i)$. When S_i is "observed" by the application, it receives $I(S_i)$ bits of information as defined in (1). $I(S_i)$ is also known as the *information entropy* of the specific input event.

$$I(S_i) = \log_2 \frac{1}{P(S_i)} \tag{1}$$

Consider a sequence of events $(S_1, S_2, S_3, \dots S_n)$ generated by an input device. If these events are independent, *i.e.*

$$P(S_n, S_{n+1}) = P(S_n)(S_{n+1}) \tag{2}$$

then we call this device a *non-memory information source*. Output symbols of keyboards, and most grammar-based recognizers, fall into this category.

Imagine a two-button keypad with a probability of 0.2 for pressing button '0' and 0.8 for pressing button '1'. Thus the information entropy associated with the event related with button '0' is log(1/0.2) = 2.32 bits and the entropy related to button '1' is log(1/0.8) =0.322 bits.

Since each event generated by an information source may occur with a different probability, each event contains a different amount of *information entropy*. Thus, the average entropy of an input device can be used to model its bandwidth or capacity. For input devices modeled as a non-memory information source, the average amount of information contained per input event is:

$$H(S) = \sum_E P(S_i)I(S_i) \quad \text{bits} \tag{3}$$

This quantity, the average amount of information per source symbol, is called the information entropy H(S) of the *non-memory information source*. H(S) will have maximum value *if and only if* each symbol is equally likely to occur (the symbols are *independent)*. Thus, to maximize the information capacity of a customized keypad, it is important to ensure that the keystrokes are as evenly distributed as possible across the vocabulary used.

For example, the average information entropy of the two button keypad example shown above is 0.2*log(1/0.2) + 0.8*log(1/0.8) = 0.722 bits. However if the two keys have an equal distribution (each has a probability of 0.5 of being used), the average entropy will increase to 0.5*log(1/0.5) + 0.5*log(1/0.5) = 1 bit. This result can be proven formally using Jensen's inequality [9].

However, in some cases the independent event condition cannot be satisfied. For example, consider a sequence of mouse cursor events. Due to the physical design properties of the mouse, if the cursor is at position (10, 12), only nearby locations such as (11, 13) are possible as the next event, and a position such as (100, 100) has a probability of almost 0. Thus, the events (10, 12) and (11, 13) are dependent and contain less information than two independent events. An *m-order Markov information source* can be used to model such information sources. In an *m*-order Markov information source, the probability of each event is specified by a conditional probability of m previous events.

$$P(S_{m+1} \mid S_1, S_2, \ldots, S_m) \tag{4}$$

The average information provided by an *m*-order Markov source is calculated as follows: if an input device is in the state specified by $(S_{j1}, S_{j2}, \ldots, S_{jm})$ (*i.e.* the *m* previous symbols emitted were $S_{j1}, S_{j2}, \ldots, S_{jm}$), then the conditional probability of receiving symbol S_i as the next event is $P(S_i \mid S_{j1}, S_{j2}, \ldots, S_{jm})$. The information available if Si occurs when an input device is in state $(S_{j1}, S_{j2}, \ldots, S_{jm})$ is:

$$I(S_i \mid S_{j1}, S_{j2}, \ldots, S_{jm}) = \log \frac{1}{P(S_i \mid S_{j1}, S_{j2}, \ldots, S_{jm})} \tag{5}$$

Therefore, the average amount of information per symbol in state $(S_{j1}, S_{j2}, \ldots, S_{jm})$ is:

$$H(S \mid S_{j1}, S_{j2}, \ldots, S_{jm}) = \sum_I P(S_i \mid S_{j1}, S_{j2}, \ldots, S_{jm}) I(S_i \mid S_{j1}, S_{j2}, \ldots, S_{jm}) \tag{6}$$

Given the current state $(S_{j1}, S_{j2}, \ldots, S_{jm})$ of an *m*-order Markov source, due to the constraints of the *m* previous states, only certain states are qualified as valid next states. Thus, the transition from the current state to the next state has a higher $P(S_{m+1} \mid S_1, S_2, \ldots S_m)$ than the corresponding state transition in a non-memory information source. This means that the average amount information in an *m*-order Markov information source is less than that in the corresponding zero-memory information source.

Based on the model described above, the bandwidth of an input device can be defined in terms of the following three criteria: (1) the average information entropy of each independent event; (2) the amount of information a user may input during a specific amount of time. *i.e.* the bit-stream speed of an input device (this brings in the human factor); and (3) Whether it is a *zero-memory information source* or *m-order Markov information source*. These three criteria can be used to classify common input devices from the information theory point of view.

2.2 Substituting one input device for another

As shown above, our model supports the comparison of different input devices. It can also be used to define a relationship between two input devices that may allow substitution, which is the ultimate goal of the IAT. The relative bandwidth of the two input devices involved plays a key role in how this is done.

2.2.1 Achieving high bandwidth with a low bandwidth device

Achieving high bandwidth with a low bandwidth device is difficult because low bandwidth devices do not carry the same amount of information as high bandwidth devices. However, there are two ways to simulate a high bandwidth device using a low-bandwidth device. First, multiple low bandwidth devices may be combined (device composition). Second, multiple actions may be done with a single low bandwidth device sequentially (device duplication). This raises two questions: How many devices are needed to simulate a certain device? What is the optimal performance that can be achieved?

The event probability from such composite information sources can be represented as the joint probability of the original event sequence.

In these cases, the probability of the combined event is:

$P(S_1, S_2, S_3, S_4 \ldots)$ (device duplication)

Or

$P(S_{11}, S_{21}, S_{31}, S_{41}\ldots)$ (device composition)

The corresponding information entropy can be represented as:

$$H(S_{new}) = \sum_N H_n(S) \tag{7}$$

For one or more low-bandwidth input source devices to be adapted to a high-bandwidth destination device, the total information generated by source input devices must be greater than or equal to the information normally generated by the destination device being simulated.

$$\left(\sum_{source\_devices} \log \frac{1}{P_i(E)} \right) \geq \log \frac{1}{P_{dest\_device}(E)} \tag{8}$$

This implies that, we need at least

$$n = \left\lceil \frac{1}{P_{dest\_device}(E)} \Big/ \log \frac{1}{P_i(E)} \right\rceil \tag{9}$$

low bandwidth devices to simulate a complex input device. Thus, equation (9) can be used to answer the first question above. The answer to the second question can be answered by applying the equal-probability principle described in Section 1.1, that is, the probability distribution of the events in source device should be as evenly distributed as possible.

2.2.2 Achieving low bandwidth with a high bandwidth device

It is far simpler to use an information source with high bandwidth to simulate an information source with low bandwidth. This is done by defining a set of new events S_i = {the generation of any events in a event list with n members $S_{i1}, S_{i2}, \ldots S_{in}$}. For example, suppose that we would like to adapt a 10-key keypad to our two-button keypad. We can define two events associated with the two button-keypad as S_1 = { the appearance of any of the symbol '0', '1', '2', .., '4' from a 10-key keypad} S_2 = { the appearance any of the symbol from list '5', '6', '7', .., '9' from a

10-key keypad}. This transformation is referred to as device splitting.

The information entropy after device splitting is (assuming the events in the source event list have equal probability)

$$H(S') = H_n(S) / r \qquad (10)$$

where $r = \log_2 (N_{source} / N_{destination})$

2.3 An Example

Suppose we would like to use a two button keypad to simulate a 10-button keypad. As given in former examples, the maximum average entropy of a two button keypad is 1 bit and the maximum average entropy of the ten keypad is log(1/0.1) = 3.32 bits. By applying equation 7 and 9, we know that we need at least 4 keystrokes of the two-button keypads to simulate a 10 key keypad event when we use device duplication and we need 4 two-button keypads (8 keys total) to simulate when we use device composition. The above statements assume that both the events of source devices and the events of destination devices are evenly distributed. Please note that the saving of physical keys (8 *vs.* 10) in the second case can be explained by the fact that the user can press keys in different devices simultaneously, thus adding more expressiveness of the input events.

3. THE INPUT ADAPTATION TOOLKIT

Based on our mathematical model, we created the Input Adaptation Toolkit (IAT) to help designers and programmers build input adaptation-related applications. In this section, we first introduce the general architecture of IAT, and then describe each module of IAT in detail.

IAT is written in Visual C++ 6.0 on Microsoft Windows 2000/XP. Figure 2 shows the architecture of IAT. IAT consists of four major components – *Input Adaptor*, *Output Exporter*, *Transformation Primitive*s and *Mapping Scheme*. The Input Adaptor receives input events from input devices and transforms them to a uniform state space. The Output Exporter generates input events that are compatible with events generated by the device to be simulated. The Transformation Primitives module supports splitting, composition, and duplication to map between devices of differing bandwidths. The Mapping Scheme encodes a one to one correspondence between the transformed state space of the source device and the state space of the destination device.

The Input adaptation process can be conceptually divided into three stages. At the stage closest to the user, user input events (created with the source device) are received by IAT via the Input Adaptor. These events may be events from a physical device (keyboard, mouse) or a software process (such as a speech recognition engine or a handwriting recognition engine). The Input Adaptor transforms the input events to a state, S_i in a uniform state space with equivalent information entropy.

At the middle stage, S_i is transformed in the Transformation Primitives module. Based on developer specifications, the Transformation Primitives module maps S_i into a state space adjusted to match the state space of the destination device. And the Mapping Scheme then converts it to a state in the state space of the destination device.

At the last stage, the Output Exporter synthesizes the destination state to one or more input events identical to one that the destination device might have generated. This is passed to the
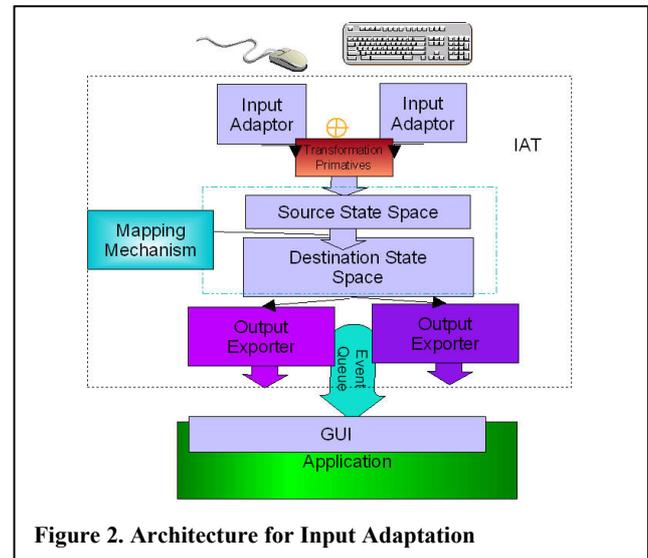

Figure 2. Architecture for Input Adaptation

application, which cannot distinguish it from events created directly by the destination device.

To adapt one device (or multiple devices) to another, a designer needs to complete two steps – (1) determine the composition topology of the Transformation Primitives in the state space; and (2) specify or create a mapping scheme.

If the source input device is not supported by IAT, the designer also needs to implement an Input Adaptor for that device. Similarly, the designer must implement an Output Exporter if the destination/simulated input device is not currently supported.

3.1 Input Adaptor

Each input device has a corresponding Input Adaptor class derived from a base class. IAT currently provides Input Adaptors for the keyboard and mouse. An Input Adaptor intercepts desired input events from a specific input device and translates those events to a uniform state space format.
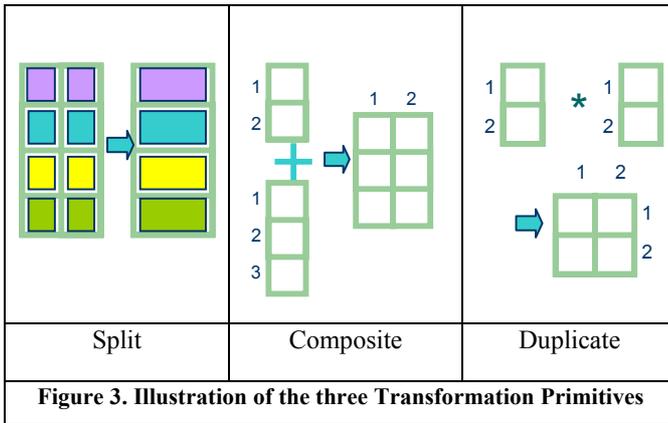
To create a new Input Adaptor, the developer must define methods that transform input events from a specific device to the standard state space format used by the mapping schemes. Similar to the Command Object presented by Myers and Kosbie [12], we provide a uniform interface for different kinds of input events. Additionally, in the case of a new piece of hardware, the developer must define the interface for intercepting input device events from the event queue of the Operating System to IAT. This interface is kept separate from the main toolkit to ensure device and operating system independence for the majority of the code.

3.2 Transformation Primitives and Mapping Scheme

IAT supports three kinds of Transformation Primitives (split, composition, and duplication, illustrated in Figure 3). They are implemented as methods of a base class that may be subclassed. By using the `Compose()` and `Duplicate()` methods, the developer can generate a larger state space from small state

spaces. By using the `Split()` method, the developer can create a smaller state space from a larger one. These three primitives can be cascaded multiple times to create complex state spaces.

The default behavior of `Split()` will divide the state space into *n* equally sized groups of adjacent elements (adjacency is determined by a system-provided iterator class, *n* is the size of the destination state space). Similarly, the default behaviors of `Compose()` and `Duplicate()` methods first treat events from source states as combination states (*e.g.* (s1, s2) for the composition of source state spaces s1 and s2) and then use a column first order to serialize the combination states (*e.g.* (0, 0) -> (0, 1) -> …-> (0, 5) …->(1, 0)) Typically, the developer will override these default behaviors.

| Split | Composite | Duplicate |

Figure 3. Illustration of the three Transformation Primitives

The Mapping Scheme module specifies the one-to-one correspondence between two state spaces of the same size. When mapping one internal state space to another, especially from a state space containing multiple low-bandwidth input devices, not only intuitive mapping rules but also visual guidelines are necessary to achieve good usability. IAT supports two methods for specifying the state space mapping.

3.2.1 State Mapping Description Language
The first mapping method uses a description language provided with IAT, the State Mapping Description Language (SMDL). SMDL is used to specify the relationship between each source state to each destination state. The current SMDL implementation supports zero-memory information sources.

SMDL mappings are specified in text format, which IAT compiles to generate a compact binary description file. The compiler verifies the syntax of the input file, organizes logically adjacent states into a Trie Tree structure and determines potential state space conflicts. The mapping class in IAT loads the compiled mapping scheme file dynamically. A mapping file consists of a description specifying high-level details about the source and destination state spaces, followed by a list of each mapping from a source event to a destination event in a one to one manner (wild-card characters can be used to specify mappings from multiple similar sources, or to multiple similar destinations). Table 1 shows part of the SMDL file used to define the mapping from a 2-button keypad to a 10 button numeric keypad by using device duplication.

```
[general definitions start here]

+AdaptationName = 2 Button

+SourceState = 01, duplication

+DestinationState = 0123456789

<..>

+Mediation = 1, 0

+AutomaticConfirmation = Y

[state mapping descriptions starts here]

00001    1

00010    2

00011    3

<..>
```

Table 1. A SMDL file describing the mapping from a 2-button keyboard to a 10 key keypad

3.2.2 Mapping Plug-In
SDML comes with some limitations. In addition to only modeling zero-memory information sources, it lacks the expressive power to describe mappings that change dynamically according to application context, or are based on heuristics or rules. Because of this, we provide a "plug-in" mechanism that, while it requires writing code, is more powerful than SDML.

A plug-in class is implemented by extending a base class provided with IAT. IAT passes an application context class to the plug-in that can be used to access the data structure and existing objects in the mapping application. Additionally, it invokes a plug-in method called `OnNewMessage()` each time the content of the Input Adaptor, Output Exporter or any state-related classes changes, or any new input events arrive. A plug-in must also provide a method that does the actual mapping. This method, `DoMap()`, is called whenever a source state should be mapped to a destination state.

3.2.3 Dealing with Ambiguities
When multiple destinations states are possible according to the current input, IAT supports mediators to let users choose from a list of candidates. Here mediation means the process of selecting the correct interpretation of the user's input [11]. A mediator is created by deriving from a base class. To save time for the developer, IAT also provides one choice-based default mediator (See Figure 4). This can be replaced or overridden when necessary.

3.3 Output Exporter
Each simulated device has a related Output Exporter class derived from a base class. IAT currently provides Output Exporters for devices that emit compatible mouse or keyboard events (*e.g.* mouse, touch screen, keyboard, keypad and speech/handwriting recognizer). An Output Exporter is responsible for creating events corresponding to the destination input device, for consumption by the application. It creates an instance of the corresponding device-specific event based on the internal state received. Then the Output Exporter synthesizes a simulated input event for the

application at the Operating System level depending on the device specific event created. All operating-system dependent code is confined to the method responsible for synthesis. In the current implementation, we use the `SendInput()` function in the Win32 API to simulate keyboard and mouse events, and MS Detours to simulate sounds.

4. APPLICATIONS BUILT WITH IAT

In this section, we briefly describe four applications built using IAT, their high-level architectures, and how input events are mapped in each.

The applications were chosen to demonstrate the capabilities of IAT. The first and simplest application, the Duo keypad, demonstrates the use of SMDL to create a mapping between two keypads of differing sizes. The second application, a re-implementation of a commercial product (T9) demonstrates the power of SMDL, and the use of customized mediators. The third application, the virtual keypad, demonstrates the use of the plug-in interface, and the use of splitting to map a high bandwidth device (the mouse) to a lower bandwidth device (a keyboard). The final application demonstrates the use of device duplication to map a low bandwidth device (a small keypad) to a higher bandwidth device (a mouse). All of these applications work with any software running in Windows.
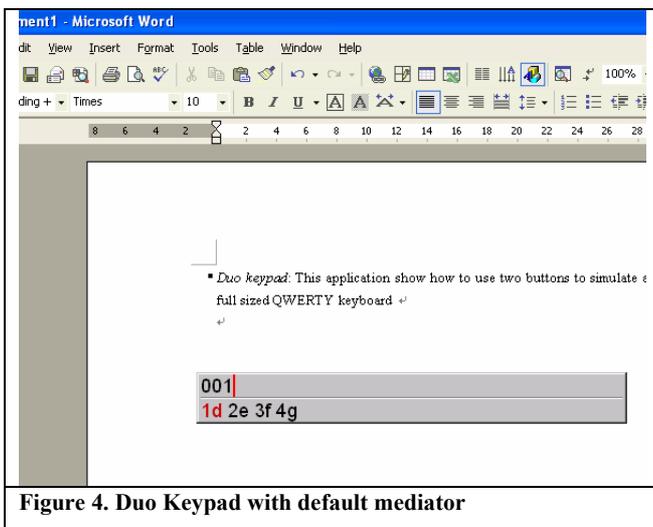
4.1 Duo keypad

Figure 4. Duo Keypad with default mediator

The Duo Keypad (Figure 4) provides full text input ability via simple and low bandwidth input devices (in this case two numeric buttons, '0' and '1'). This application demonstrates using a low bandwidth device (1 bit per key) to simulate a high bandwidth input device (4.7 bits per key) through device duplication.

Duo Keypad uses the default Keyboard Input Adaptor and keyboard Output Exporter provided by the IAT. An SMDL file is used to define the rules for mapping the source input device to the destination input device. A default mediator from IAT displays potential candidates when there are ambiguities. No additional coding is necessary to implement the Duo Keypad application.

4.2 T9 input method

The T9 input method is a widely used predictive text entry method design for small devices such as cell phone or PDAs [7]. T9 enables a user to enter English sentences via a keypad with almost the same number of keystrokes as the full sized QWERTY keyboard by exploiting dictionary knowledge. In this application, the source input device is a 10 button keypad with an average information entropy of 3.32. The destination device is no longer a keyboard, but a virtual device with words as potential output events. If the word list includes 1000 words, the maximum average information entropy per event is I(word) = 12.55. As a result, in the optimal cases, we need to duplicate a 10-button keypad four times to simulate the destination virtual device. The word prediction performance has an upper bound of I(word)/I(key), where I(k) is the maximum average information entropy per key stroke, which from our previous calculations is 3.32. Thus, T9 with 1000 words is bounded by 12.55/3.32 = 3.89 keystrokes per word, assuming all words are equally likely.
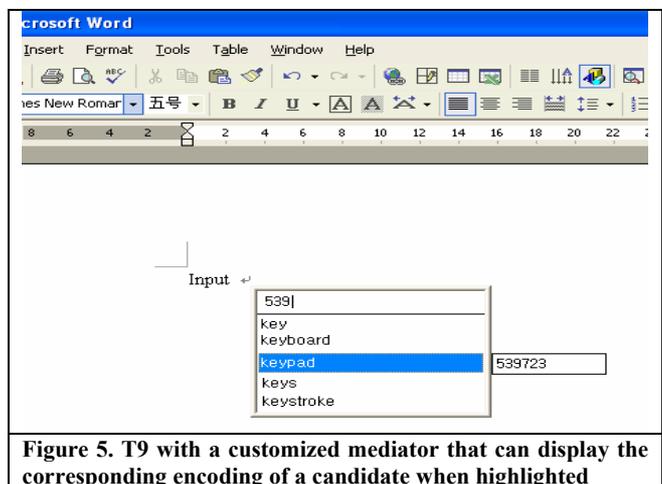

Figure 5. T9 with a customized mediator that can display the corresponding encoding of a candidate when highlighted

Implementing T9 using IAT is straightforward. The easiest method is to use SMDL. This requires each supported word to be specified. For example, the digit sequence '539729' maps to the word 'keypad,' so there is one entry in the SMDL file specifying this mapping.

The demonstration application we implemented supports over 6000 words. Although the mapping file is large, since the source states are reorganized in Trie tree structure, after the compilation, the throughput is about 1.5 million words per minute on a Pentium III 1G Laptop, easily fast enough to meet human performance needs. Figure 5 shows the T9 text entry application. Note the customized mediator, which differs from that shown in Figure 4. Although not required for T9 to work, this new mediator enhances usability by showing the corresponding T9 encoding when a candidate word is highlighted.

One limitation of the SMDL T9 implementation is that if the word a user intends to enter is not in T9's word list, the user can not enter the word. This can be overcome by defining an alternative multi-tap input list, such as 2 -> A, 22->B 222->C. When the intended word is not in the word list, the user can switch the SMDL file dynamically to multi-tap and input that word at the

character level. Another possible solution is to add a customized mediation method to resolve the character ambiguity.

4.3 Virtual Keyboard

The virtual keyboard application (Figure 6) shows how to simulate a relatively low bandwidth 115 key QWERTY keyboard (6.85 bits per keystroke) with a high bandwidth mouse (around 20 bits per random click).

The virtual keyboard application also demonstrates two features of IAT. First, the plug-in interface is used to create the window of the virtual keyboard and to define the mapping from source states (mouse events) to destination states (keyboard events). Since the mapping from mouse position to keyboard depends on the current position of virtual keyboard window, it cannot be defined using SMDL. When a mouse event is intercepted, the plugin checks whether it falls into the virtual keyboard window. If it does, the source state space is split into the destination state space by determining some pre-calculated state boundaries reflecting the on-screen key positions. Then related keyboard events are generated based on the destination state space. The control, shift, and caps lock key are handled specially by updating three pre-defined variables associated with them.
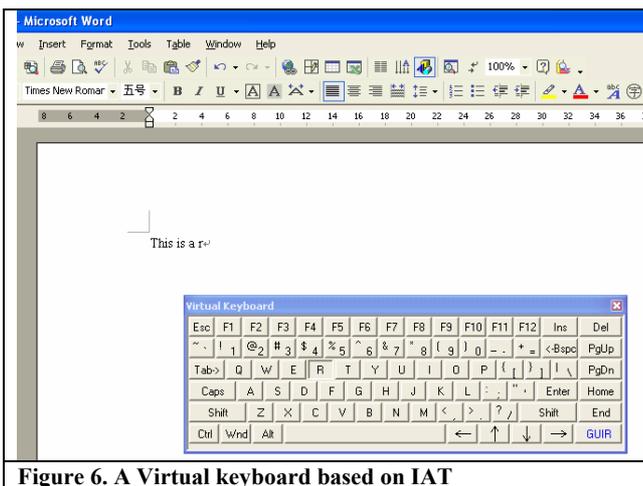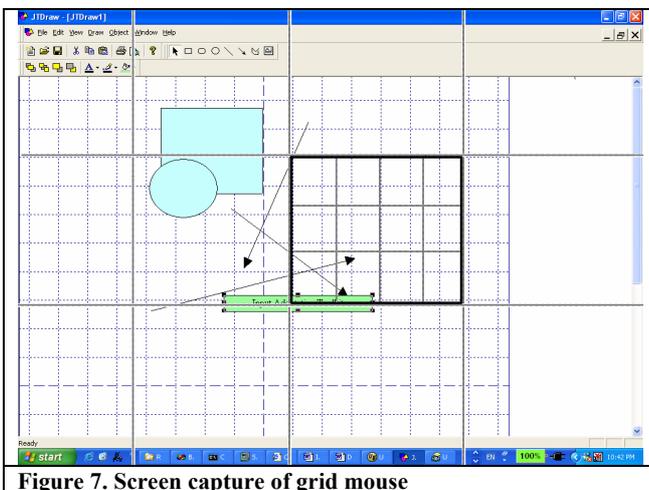

Figure 6. A Virtual keyboard based on IAT

Figure 7. Screen capture of grid mouse

4.4 Grid Mouse

The Grid mouse (Figure 7) demonstrates the use of four arrow keys and a space bar (2.1 bits per keystroke) to simulate a pointing device (around 20 bits per random event). Note that the Grid mouse does not support dragging, but can be used to select an arbitrary position on the screen.

Because of the difference in bandwidth, up to 10 key presses are needed to move the mouse cursor to a random position of the screen. Device duplication is used to do this. Additionally, the Grid mouse application demonstrates the use of the plug-in to create visual guidelines to guide the user's input.

5. RELATED WORK

Two categories of existing research are directly related to our contributions. First we discuss past work in input device modeling. Second, we discuss past work in user interface adaptation.

Several researchers have done seminal work in modeling input devices and the input process. Buxton developed a three-state state machine that could model a variety of pointing devices such as mice and touch screens [3]. The three states used in the model are *tracking*, *dragging* and *out of range*. Rudnicky and Hauptmann also present a state-machine model of input devices that focuses on modeling error correction [16]. Hudson *et al.* introduced Probabilistic State Machines that extend input events to carry information about uncertainty originating from recognition based input devices [8]. That work focused on the event handling stage of an interface toolkit. A probabilistic state machine represents an input device as a state machine, but rather than keeping track of a single state, it updates all of the possible states in a probabilistic manner. Card and colleagues presented a morphological design space to systematize the capacity and internal working mechanisms of input devices [6]. In their characterization, an input device can be composed of basic input devices *via* composition operators. Factors such as whether a device is rotated, moved, or pressed and relative *versus* absolute positioning were used to describe the input devices. This model could not easily describe a software device such as a speech recognizer or soft keyboard. The authors used Fitts' Law and other modeling to predict the human performance speed for a specific task.

In summary, these existing models focus on the formalization of the internal working mechanisms or mechanical composition of input devices. However, none of them models the relative bandwidth capacity of input devices, and none of them explicitly include non-physical devices such as a speech recognizer.

In terms of user interface adaptation, user interface transformation is perhaps the approach most closely related to IAT. Typically, this involves transforming a GUI or web page originally designed for a desktop-sized display to fit on a small screen device [2] [3][5], or rendering a GUI in a new modality [13]. An alternative is to define an interface in platform-independent terms. For example, the XWeb architecture supports cross model interaction using this approach [15], as does the User Interface Markup Language (UIML) [18] Other frequently adopted approaches to user interface transformation include using a proxy (usually for web related applications) [3][10], and using hook functions to intercept system display related function calls [15][14] or to display an interface in a different modality [13], and adding "glue

codes" to the original application and then calling some runtime transformation library [17].

All of the methods described above are focused on changing the "look and feel" of the user interface in some way, with changes to input being an additional feature in some cases [13][15]. Researchers have paid little attention to providing a uniform method for input device transformation. In other words, past work has focused on the "output" part of adaptation while our approach is dedicated to the "input" part of the adaptation.

6. Conclusions and Future Work

To accommodate the needs of a diverse user population, it is important that applications be able to accept input from input devices for which they were not designed. This is particularly important for users with severe motor impairments, who may use very low-bandwidth input devices. In addition, blind users and users of PDAs, cellphones, *etc.* all have differing input needs from those assumed by current desktop computing applications.

We have presented a model of input devices that defines their relative bandwidth based on the set of events, or *states,* associated with them. Our model can be applied in the design and analysis of input device translations. We went further, and used the model to design and create a toolkit for translating between input devices. We demonstrated our tool with four applications of varying complexity. Two can be built without coding, and two require developer coding. All can run in Windows with any desktop application. This is a work in progress, and we plan to continue development on IAT to increase its expressive power and decrease the complexity of writing translation code. In particular, we plan to extend the grammar of SMDL to support *m*-order Markov information sources, and to enhance the set of mediators available.

Most past work has focused on the problem of converting output from one device to another. In contrast, our tool solves the problem of translating between input devices. In the future, we hope to combine our tool with an output adaptation tool, to provide a complete solution for adaptation needs on both sides.

ACKNOWLEDGMENTS

REFERENCES

[1] Abramson, N. Information Theory and Coding, McGraw-Hill Inc, 1963.

[2] Avantgo Mobile Channel. http://www.avantgo.com

[3] Barrett, R., and Maglio, P. P., Intermediaries: New places for producing and manipulating web content. In *Proc. of WWW7*, pp. 509-518. 1998.

[4] Buxton, W. A., A Three-state Model of Graphical Input. Human-Computer Interaction. In *Proc. of INTERACT '90.* pp. 449-456, 1990.

[5] Buyukkokten, O., Garcia-Molina, H., Paepcke, A., and Winograd, T., Power browser: efficient Web browsing for PDAs. In *Proc. of CHI'00,* pp. 430-437. 2000.

[6] Card, S., Mackinlay, J., and Robertson, G., A Morphological Analysis of the Design Space of Input Devices. In *ACM Transactions on Information Systems*, **9**(2):99-122. April 1991.

[7] Grover, D. L., King, M. T., and Kuschler, C. A., Patent No. US5818437, *Reduced keyboard disambiguating computer.* Tegic Communications, Inc., Seattle, WA 1998.

[8] Hudson, S., and Newell, G., Probabilistic State Machines: Dialog Management for Inputs with Uncertainty. In *Proc. of UIST '92,* pp.199-208. 1992

[9] Krantz, S. G. "Jensen's Inequality", in *Handbook of Complex Analysis*. Birkhäuser. p. 118. 1999.

[10] Mankoff, J., Dey, A., Batra, U., and Moore, M., Web Accessibility for Low Bandwidth Input. In *Proc. of ASSETS 2002*, pp. 17-24. 2002.

[11] Mankoff, J., Hudson S., and Abowd, G., Providing Integrated Toolkit-Level Support for Ambiguity in Recognition-Based Interfaces. In *Proc. of CHI 2000*, pp. 368-375. 2000.

[12] Myers, B. and Kosbie, D., Reusable Hierarchical Command Objects. In *Proc. of CHI '96*, pp. 260-267. 1996.

[13] Mynatt, E.D., and Edwards, W. K., Mapping GUIs to Auditory Interfaces, In *Proc. of UIST '92*, pp. 61-70. 1992

[14] Olsen, D., Hudson, S., Verratti, T., Heiner, J., and Phelps, M., Implementing interface attachments based on surface representations. In *Proc. of CHI '99,* pp. 191-198. 1999.

[15] Olsen, D. R., Jefferies, S., Nielsen, T., Moyes, W. and Fredrickson, P., Cross-modal interaction using Xweb. In *Proc. of UIST 2000,* pp. 191-200. 2000.

[16] Rudniky, A., and Hauptmann, A., Models for evaluating interaction protocols in speech recognition. In *Proc. of CHI'91,* pp. 285-291. 1991.

[17] Smith, I. E., *Support for Multi-Viewed Interfaces*. Ph.D. Thesis, Georgia Institute of Technology, 1998.

[18] User Interface Markup Language 3.0 http://www.uiml.org/specs/