ButtonSchemer: Ambient Program Reader

Nwanua Elumeze
Dept. of Computer Science
University of Colorado
Boulder, CO 80309 USA
nwanua@cs.colorado.edu

Michael Eisenberg
Dept. of Computer Science
University of Colorado
Boulder, CO 80309 USA
duck@cs.colorado.edu

ABSTRACT
This paper describes ButtonSchemer, an "ambient program reader" that can be used to input program code directly from a computer screen or from specially bar-coded surfaces. The placement of programs for such a device can be made informal, creative, and practically ubiquitous, suggestive of ways to extend the traditional desktop-centric notions of programming.

Categories and Subject Descriptors
K.8.m [Personal Computing]: Miscellaneous.

General Terms
Design

Keywords
Children's programming; ambient programming; computational crafts; wearable computing; electronic textiles; Scheme.

1. INTRODUCTION
Over the past decade, human-computer interaction has increasingly focused on the broad area of ambient computing. Certainly, as computation is embedded within an ever-larger array of physical artifacts over time, the range of scenarios for interaction likewise increases. Still, the overwhelming majority of these scenarios for ambient computing exhibit an intriguing "blind spot"—they envision a world in which users carry computers, interact with ubiquitous computers, or even wear computers, but never actually write programs for those computers themselves.

That is, the ambient computing community seems to pay little attention to the possibility of ambient programming—the ability of users to create and input programs appropriate to the burgeoning world of ambient computational devices. Just as computers need no longer sit on a desktop, it is likewise the case that programming—informally, "in the small"—can take place in a variety of settings and physical spaces as well.

In short, the advent of embeddable computation suggests a new and potentially quite powerful means to challenge, complement, and extend the traditional desktop-centric notions not only of computing, but of programming as well.

This paper describes a novel computational artifact called ButtonSchemer, whose purpose is to explore and extend the possibilities for programming away from the desktop. Briefly, ButtonSchemer (see Figure 1) is a small device that can be sewn onto fabric, as an element of a computationally-enriched garment or textile. The device is an "ambient program reader" that can be used to input program code directly from a computer screen by being placed in front of the screen (i.e., the reader does not use a wire-based connection). Alternatively, one can read programs into ButtonSchemer by sweeping it physically across appropriately marked materials such as paper or cloth, in the manner of a barcode reader. One might want to think of ButtonSchemer overall, therefore, as a kind of "program-reading button", sewn onto something like a sleeve or bracelet. Once the device has read in a program, that program can be used to activate lights, motors, or buzzers; the device may also be employed to read other input besides programs (such as sensor values).

In Figure 1, we provide an example instantiation of a ButtonSchemer. A device such as ButtonSchemer thus represents an approach toward ambient, day-to-day programming. One might use the device to read in a program directly from the screen of a handheld computer or laptop; or one might sweep one's ButtonSchemer-equipped bracelet along a surface on which an appropriately encoded program has been written (in a manner to be described below). The upshot of this design is that one might read in programs from all sorts of serendipitous sources—a friend's laptop, a "program mural" on a wall, a printed-out pattern on paper—to customize (say) a computational wearable. A ButtonSchemer program can thus show up in all sorts of creative and unexpected ways.
places in one’s environment: in the final section of this paper we will pursue this idea by presenting a few plausible scenarios.

2. RELATED WORK
ButtonSchemer builds upon, and has been influenced by, a tradition of research in creating learnable programming environments for small or embeddable computers; in many cases, these environments are aimed at hobbyists or children as a presumed audience. Much of the early work in this area derived from a hobby culture in robotics—often, small computers were used to control model creatures or vehicles built from materials such as Lego.

When users first began to program embedded computers, they needed to go through a cumbersome series of steps: detaching the portion of the device (chip) containing code, plugging it into an expensive and specialized EPROM programmer, and then finally replacing it. During the mid-90’s, as flash-based non-volatile memory became increasingly available, it became possible for users to program an embedded computer in place, without physically moving any chips. This led to a proliferation of research and commercial products that quickly enabled hobbyists to create all sorts of computational artifacts.

The landscape of "small computer programmability" has thus become more varied over recent years. Some products are designed with at least an emphasis upon robotics—Lego Mindstorms [6] is a good example of this genre. Other products, like the Arduino [2] are more general purpose and require the user to provide their own sensors and actuators. The recent Picocricket [9] comes with a set of "intelligent" input and output devices that simplify wiring and programming; while, in a different vein, the Lilypad [7] interweaves computation with electronic textiles.

Some of these systems come with a dedicated programming language, development environment, and hardware—and each has a community of adherents. Still, the task of programming physical computational artifacts remains a challenge. Part of this challenge stems in part from an issue these projects have in common: they require the user to install special software, purchase additional hardware, and make special connections to a computer in order to get started. With more general-purpose programming platforms, users are faced with a frustrating variety of software conflicts, inconsistent implementations, incoherent error messages, and poor documentation. Some users do manage to overcome these various challenges; but undoubtedly many others find the task of programming an embedded computer too daunting even to make the attempt.

3. PLATFORM
ButtonSchemer addresses these issues in two ways. First, it can be programmed in an extremely simple fashion, merely by holding it in front of a computer screen (Figure 2): specially timed flashes within a small box on the screen convey programming bits that are received by an onboard light sensor. No special or extra equipment is needed, just the display itself. ButtonSchemer can also be programmed when swept along a surface containing appropriately encoded bar codes.

Second, ButtonSchemer’s compiler and programmer elements are written entirely in JavaScript, and can be run from any standard browser. This eliminates (or at least minimizes) installation problems, as the user needs only access one single HTML file. See [nwana.aniomagic.com/buttonschemer.html] for a demonstration movie.

To run user programs, ButtonSchemer makes use of a built-in, rudimentary Scheme interpreter that executes the bytecodes produced by the browser-based compiler (or represented in the bar-codes). Scheme[1] is a small yet powerful high-level programming language that nicely provides tail recursion, block structure and lexical scoping. It is also remarkable in that a variety of object types have “first-class” status, (e.g. procedures, like data, can be given names and employed freely as arguments, returned values, or list elements).

![Figure 2. Screen Programming with the ButtonSchemer](image)

By convention, ButtonSchemer runs a procedure named **main**: thus, a user who wishes to program the device writes a Scheme procedure with this name and sends it to the device. Normally, the program would loop continuously, unless the programmer specifies otherwise. The ButtonSchemer language also includes a special primitive procedure named **button-read**, which pauses a running program and waits for the device to read in a representation of a Scheme object (which can be a number, procedure, symbol, list, or text string, among other possibilities).

To illustrate, assume the user sent the following to ButtonSchemer:

```scheme
(define (main)
  (let ((repetitions (button-read)))
    (repeat repetitions (flash light1)))
)
```

After reading in this new **main** procedure (and replacing the old one with it), ButtonSchemer waits at **button-read** for a value. If it is physically passed over a bar code representing an integer (say 4), the light will flash 4 times.

In another example, the user initially sends:

```scheme
(define (main)
  (let ((procedure-to-use (button-read)))
    (repeat (procedure-to-use 3)))
)
```
Subsequently, to cause an attached motor to spin 270 degrees, one might send:

(procedure (n) (clockwise-turn motor-a (* n 90)))

The point of these examples is merely to show that one can write ButtonSchemer programs that read in integers, lists, procedures, or other types of objects. The only constraint (consistent with all Scheme programming) is that these read-in objects must be used appropriately within the surrounding program—e.g., one cannot try to take the numeric sum of a number and a procedure.

The small size of user-accessible memory means current programs must be short. Our browser-based compiler aggressively compacts object representation [5], and the onboard ButtonSchemer bytecode interpreter uses a limited version of the mark-sweep garbage-collecting algorithm. The system implements only a skeletal Scheme environment, with many standard language elements omitted. The low bandwidth method of flashing the screen means programs must be short: it takes about 1 minute to transmit 64 bytes.

Nevertheless, we’ve found that even simple programs can generate delightful effects when they control the behavior of physical objects like a computational garment. Moreover, it should be noted that one can write ButtonSchemer programs that respond to external sensor input; thus, the unpredictability of the surrounding environment might cause even a simple program to behave in surprising ways.

4. FUTURE SCENARIOS

The notion of ambient programming suggests the possibility of exploring novel ways of presenting programming ideas in day-to-day settings. We envisage a number of exciting scenarios along these lines. Though still a bit futuristic, the scenarios discussed here illustrate some potential medium-term directions for development of the ButtonSchemer platform.

Suppose, then, that a child has a ButtonSchemer-equipped bracelet that includes (as output elements) six multi-color lights and (as input sensor element) a pushbutton switch. In a more traditional paradigm of programming, if she wanted the lights to cycle through a list of colors, she would need to write some kind of textual program at a computer, connect the artifact to that computer, and download the code.

Instead, because ButtonSchemer can plausibly read programs written in barcode form, in this scenario the child has a deck of specially designed cards with barcodes on them. The cards, in effect, embody a special-purpose “iconic programming dialect” in which the textual elements are encoded underneath a pictorial representation. To read in a program or command, the child sweeps her bracelet over the barcode elements of a card or sequence of cards.

We imagine, then, that the child could read in a list of colors to her bracelet by laying out a “make-a-named-list” card, followed by several color cards: (e.g. red, purple, green, as shown in Figure 3). She then scans the bracelet over the sequence of cards.

Next, she lays down a LIGHT card, followed by the same LIST card in the previous step, and swipes her bracelet again. The effect here is to tell the LIGHT card to use the named list as the argument to a procedure that runs through a sequence of colors.

Now her bracelet would continuously cycle "red, purple, green, red, purple, green..." If at some point she wanted to change the colors or their order, she would do so by creating another list of colors (similar to Figure 3). Different LIST cards refer to different lists, and she may use them in any combination.

We could pursue this example a bit further. Suppose the child wants to alter her program to cycle to the next color in response to a sensor input—say, only when a switch is pressed, or when it’s warm enough, or very quiet, or when she swings her arm rapidly. She need only insert the appropriate sensor card between the LIGHT and LIST cards:

Now she would like to make changes to the light only when she has pressed the switch 4 times. Perhaps she uses each tap to keep track of events. Now she might include a bit of textual language to augment the deck of cards:
The purpose of these examples is merely to suggest that, through ambient programming, we can envision strategies for bridging the traditional dichotomy between expressive, textual programming languages that can represent abstraction and recursion; and graphical/physical languages that of necessity, remain simple. By blending textual and pictorial elements that can be read into an ambient device, it should be possible to create expressive, short programs in a way not previously feasible away from the desktop.

As the prospect of truly ambient computing becomes more attainable, there has been a corresponding increase in Human Computer Interaction (HCI) research on how we might interact with these computers.

There are still other scenarios that are suggested by the advent of devices like ButtonSchemer. One could make a truly "ubiquitous" program by writing it on a desktop computer, printing it (with the corresponding bar-codes) on sheets of paper, and pasting the sheets on all sorts of surfaces. Casual passersby can see and read-in programs thus scattered through an environment. One could also acquire a collection of procedures for later use/modification in this manner. With crayons, fabric paint, or markers, one could even “write”, by hand, short programs—transcribing a sequence of bar codes on walls, clothing, bags, or a favorite book cover.

One might bring an ambient ButtonSchemer to an ambient program; or, alternatively, one could carry a program directly to some ambient ButtonSchemer device. One might walk up to a ButtonSchemer-equipped mobile, wall (e.g. SmartTiles[4]), or shirt (e.g. Buechley’s Tanktop[3]), and pass a series of cards over it in order to customize the object’s running program. This permits a rethinking of the notion of “ambience” itself: in these examples, the program becomes ubiquitous, even if the computation itself remains attached to (say) a building.

The existing platform and language also allows entire procedures and lists of data to be passed between computational artifacts. For example, a child’s shirt could be programmed to “flash” a procedure to a ButtonSchemer-equipped classroom wall, in order to (for instance) take advantage of the wall’s more powerful computational resources. Results and data could then easily be sent back to the shirt.

The current prototypes offer a very promising proof of concept, but there are issues to work out, including limited program size, requirement for sufficient light, as well as adequate bandwidth for programming. There are also questions about losing line of sight as the reader must be held directly in front of the screen.

Being able to print out a program on paper, share it with friends, or physically pass it over others’ computational artifacts enables us to tastefully extend the traditional landscape of novice programming into unexpected, informal settings. While the ButtonSchemer work is still at an early stage, we believe that it hints at a much broader notion of “ambient computation” than has been traditionally purveyed.

5. ACKNOWLEDGMENTS
Thanks to Mark Gross and members of Craft Technology Lab. This research was funded in part by the National Science Foundation (awards no. EIA-0326054, and REC0125363).

6. REFERENCES
[7] Lilypad. cs.colorado.edu/~buechley/diy/diy_lilypad_arduino.html