User Interface Design With Matrix Algebra

HAROLD THIMBLEBY
UCLIC, University College London Interaction Centre

It is usually very hard, both for designers and users, to reason reliably about user interfaces. This article shows that 'push button' and 'point and click' user interfaces are algebraic structures. Users effectively do algebra when they interact, and therefore we can be precise about some important design issues and issues of usability. Matrix algebra, in particular, is useful for explicit calculation and for proof of various user interface properties.

With matrix algebra, we are able to undertake with ease unusally thorough reviews of real user interfaces: this article examines a mobile phone, a handheld calculator and a digital multimeter as case studies, and draws general conclusions about the approach and its relevance to design.

Categories and Subject Descriptors: B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*User interfaces*; H.1.2 [**Models and Principles**]: User/Machine Systems; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Theory and methods*

General Terms: Design, Human Factors, Performance

Additional Key Words and Phrases: Matrix algebra, usability analysis, feature interaction, user interface design

> "It is no paradox to say that in our most theoretical moods we may be nearest to our most practical applications." *A. N. Whitehead*

1. INTRODUCTION

User interface design is difficult, and in particular it is very hard to reason through the meanings of all the things a user can do, in all their many combinations. Typically, real designs are not completely worked out and, as a result, very often user interfaces have quirky features that interact in awkward ways. Detailed and precise critiques of user interfaces are rare, and very little knowledge in design generalizes beyond specific case studies. This article addresses these problems by showing how matrix algebra can be applied to user interface design. The article explains the theory in detail and shows it applied to three real case studies.

Author's address: UCLIC, University College, London, Remax House, 31/32 Alfred Place, London, WC1E 7DP, United Kingdom; email: h.thimbleby@ucl.ac.uk; URL: http://www.uclic.ac.uk.

Push button devices are ubiquitous: mobile phones, many walk-up-and-use devices (such as ticket machines and chocolate vending machines), photocopiers, cameras, and so on are all examples. Many safety critical systems rely on push button user interfaces, and they can be found in aircraft flight decks, medical care units, cars, and nuclear power stations, to name but a few. Large parts of desktop graphical interfaces are effectively push button devices: menus, buttons and dialog boxes all behave as simple push button devices, though buttons are pressed via a mouse rather than directly by a finger. Touch screens turn displays into literal push button devices, and are used, for example, in many public walk-up-and-use systems. The World Wide Web is the largest example by far of any push button interface. Interaction with all these systems can be represented using matrix algebra.

Matrices have three very important properties. Matrices are standard mathematical objects, with a history going back to the nineteenth century.[1] This article is not presenting and developing yet another notation and approach, but it shows how an established and well-defined technique can be applied fruitfully to serious user interface design issues. Second, matrices are very easy to calculate with, so designers can work out user interface issues very easily, and practical design tools can be built. Finally, matrix algebra has structure and properties: designers and HCI specialists can use matrix algebra to reason about what is possible and not possible in very general ways. This article gives many examples.

There is a significant mathematical theory behind matrices, and it is drawing on this established and standard theory that is one of the major advantages of the approach.

Matrices are not necessarily device-based: there is no intrinsic 'system' or 'cognitive' bias. Matrix operations model actions that occur when user *and* system synchronize in what they are doing. Thus a matrix represents as much the system responding to a button push as the user pressing the button. Matrices can represent a system doing electronics to make things happen, or they can represent the user thinking about how things happen. The algebra does not 'look' towards the system nor towards the user. As used in this article, it simply says what is possible given the definitions; it says how humans and devices *interact*.

Readers who want a review of matrices should refer to the many textbooks available on matrix algebra (linear algebra); Broyden's [1975] *Basic Matrices* is one that emphasises partitions, a technique that is used extensively later in this article; Liebler's [2003] more modern textbook, *Basic Matrix Algebra with Algorithms and Applications*, is very accessible and covers a much wider range of matrix algebra. Readers who want a brief but mathematically formal background should refer to the Appendix of this article.

[1]The Chinese solved equations using matrices as far back as 200 BC, but the recognition of matrices as abstract mathematical structures came much later.

2. CONTRIBUTIONS TO HCI

There are many theories in HCI that predict user performance. Fitt's Law can be used to estimate mouse movement times; Hick's Law can be used to estimate decision times. Recent theories extend and develop these basic ideas into systems, such as ACT/R [Anderson and Lebiere 1998], that are psychologically sophisticated models. When suitably set up, these models can make predictions about user performance and behavior. Models can either be used in design, on the assumption that the models produce valid results, or they can be used in research, to improve the validity of the assumptions. ACT/R is only one of many approaches; it happens to be rather flexible and complex—many simpler approaches, both complementary and indeed rival have been suggested, including CCT [Kieras and Polson 1985]; UAN [Hartson et al. 1990]; GOMS [Card et al. 1983]; PUM [Young et al. 1989]; IL [Blandford and Young 1996] and so on (see Ivory and Hearst [2001] for a broad survey of evaluation tools). All these approaches have in common that they are psychologically plausible, and to that extent they can be used to calculate how users interact, for instance to estimate times to complete tasks or to calculate likely behavior. Some HCI theories, such as information foraging [Pirolli and Card 1998], have a weaker base in psychology, but their aim, too, is to make predictions of user behavior. Of course, for the models to provide useful insights, they must not only be psychologically valid but also based on accurate and reasonable models of the interactive system itself.

Unlike psychologically-based approaches, whose use requires considerable expertise, the only idea behind this article is the application of standard mathematics. The ideas can be implemented by anyone, either by hand, writing programs or, most conveniently, by using any of the widely available mathematical tools, such as Matlab, Axiom or *Mathematica* (see Section 9.3). Matrix algebra is well documented and can easily be implemented in any interactive programming environment or prototyping system. User interfaces can be built out of matrix algebra straight forwardly, and they can be run, for prototyping, production purposes, or for evaluation purposes.

An important contribution this article makes is that it shows how interaction with real systems can be analyzed and modelled rigorously, and theorems proved. One may uncover problems in a user interface design that other methods, particularly ones driven from psychological realism, would miss.

The method is simple. However, I emphasise that throughout this article we will see 'inside' matrices. This gives a misleading impression of complexity, and of the work required to use matrices effectively. The contents of a matrix and how one calculates with it can be handed over to programs; the inside details are irrelvant to designers—and, of course, inside a typical interactive program the matrices will be implemented as 'black boxes' in program code, rather than as an explicit arrays of numbers as used for clarity throughout this article. For this article, however, it is important to see that the approach works and that the calculations are valid. The danger is that this gives a misleading impression of complexity (because the calculations are explicit), whereas it is intended to give an accurate impression how the approach works, and how from a mathematical point of view it is standard and routine. Conversely, because we

have presented relatively small examples, emphasising manageable exposition despite the explicit calculations, there is an opposite danger that the approach seems only able to handle trivial examples!

2.1 Methodological Issues

This article makes a theoretical contribution to HCI. One might consider that there are two broad areas of theoretical contributions in HCI, which aspire, respectively, to psychological or computational validity. This article makes computational contributions, and its validity does not depend on doing empirical experiments but rather on its theoretical foundations. The foundations are standard mathematics and computer science, plus a theorem (see the Appendix). The theorem, once stated, seems very obvious but it appears to be a new insight, certainly for its applications to HCI and to user interface design.

The issue, then, for matrix algebra is not its psychological validity but whether the theoretical structure provides new insight into HCI. I claim it does, because it provides an unusual and revealing degree of precision when handling real interactive devices and their user interfaces.

For example, one might do ordinary empirical studies of calculator use and see how users perform. But, as I will show, there are some reasonable tasks that cannot be achieved in any sensible way—and this result is provable.

It may be established empirically whether and to what extent such impossible tasks are an issue for users under certain circumstances, but for safety critical devices, say electronic diving aids for divers, or instrumentation in aircraft flight decks, the ability or inability to perform tasks is a safety issue, regardless of whether users in empiricial experiments encounter the problems. Thus the theoretical framework raises empirical questions or raises risk issues, both of which can be addressed *because* the theory provides a framework where the issues can be discovered, defined and explored.

This article provides a whole variety of nontrivial design insights, both general approaches and those related to specific interactive products: almost all of the results are new, and some are surprising. The real contribution, though, is the simple and general method by which these results are obtained.

In proposing a new theory, we have the problem of showing its ability to scale to interesting levels of complexity. The narrative of this article necessarily covers simple examples, but I wish to imply that bigger issues can be addressed. My approach has been to start with three real devices. Being real devices, any reader of this article can go and check that I have understood them,[2] represented them faithfully in the theory, and obtained non-trivial insights. The three examples are very different, and I have handled them in different ways to illustrate the flexibility and generality of the approach. Also, I do everything with standard textbook mathematics; I have not introduced parameters or fudge factors; and I have not shied away from examining the real features and properties of the example systems.

A danger of this approach is that its accuracy relies on the accuracy of reverse engineering; the manufacturers of these devices did not provide their formal

[2]Device definitions and other material is available at `http://www.uclic.ucl.ac.uk/harold`.

specifications—I had to reconstruct them. While this is a weakness, any reader can check my results against a real device, its user manual (if adequate), or by entering into correspondence with the manufacturer. Of course, different manufacturers make different products, and in a sense it is less interesting to have a faithful model of a specific proprietary device than to have a model of a generic device of the right level of complexity; by reverse engineering, even allowing for errors, I have certainly achieved the latter goal.

An alternative might have been to build one or more new systems, and exhibit the theory used constructively. Here, there would be no doubt that the systems were accurately specified—though the article would have to devote some space to providing the specifications of these new devices (which the reader cannot obtain as physical devices). But the worse danger would be that I might not have implemented certain awkward features at all: I would then be inaccurately claiming the theory handled, say, 'mobile phones' when in fact it only handled the certain sort of mobile phone that I was able to implement. It would then be very hard to tell the difference between what I had done and what I should have done.

For this article, I have chosen relatively cheap handheld devices. Handheld devices are typically used for specific tasks. Again, this makes both replication of this work and its exposition easier. However, further work might wish to consider similar sorts of user interface in different contexts, for example, in cars—to radios, audio systems, navigation, air conditioning, security systems, cruise control and so on. Such user interfaces are ubiquitous, and typically over-complex and under-specified. Computer-controlled instruments contribute increasingly to driver satisfaction and comfort. The drivers have a primary task which is not using the interface (and from which they should not be distracted), so user interface complexities are more of an issue. Cars are high-value items with long lifetimes, and remain on the market much longer than handheld devices. They are more similar, and used by many more people. Even small contributions to user interface design in this domain would have a significant impact on safety and satisfaction for many people.

2.2 The Theory Proposed

The theory is that users do algebra, in particular (for the large class of systems considered), linear algebra. Linear algebra happens to be very easy to calculate with, so this is valuable for design and research. However, it is obviously contentious to say that users *do* algebra! We are not claiming that when people interact with systems that they engage in cognitive processes equivalent to certain specific sorts of calculation (such as matrix multiplication), but that they and the system they interact with obey the relevant laws of algebra. Indeed, if users were involved in requirements specification, they may have expressed views that are equivalent to algebraic axioms.

Users do algebra in much the same way as a user putting an apple onto a pile of apples "adds one" even if they do not do any sums in their heads. Users of apple piles will be familiar with all sorts of properties (e.g., putting an apple

in, then removing an apple leaves the same quantity; or, you cannot remove more apples from a pile than are in it; and so on) regardless of whether anyone does the calculations. Likewise, users will be (or could well be) familiar with the results of matrix algebra even if they do not do the calculations. Only a philosophical nominalist could disagree [Brown 1999] with this position.

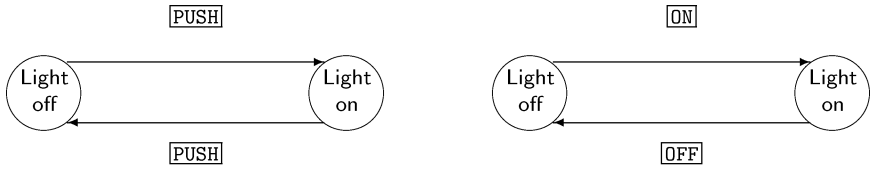
The brain is a symbolic, neural, molecular, quantum computer of some sort and there is no evidence that users think about their actions in anything remotely like calculating in a linear algebra. Yet we can still make tentative cognitive claims. There is no known easy way to invert a matrix. Therefore if a user interface effectively involves reasoning about inverses, it is practically certain that users will find it difficult to use reliably. Or: A user's task can be expressed as a matrix; thus performing a task/action mapping (going from the task description to how it is to be achieved) is equivalent to factoring the matrix with the specific matrices available at the user interface. Factorization is not easy *however* it is done (although there are special cases)—we can conclude that whatever users do psychologically, if it is equivalent to factorization, it cannot be easy (even if they have tricks and short cuts). Indeed, we know that users keep track of very little in their heads [Payne 1991], so users are likely to find these operations harder, not easier than the 'raw' algebra suggests.

What we do know about psychological processes suggests that the sort of algebraic theory discussed in this article is never going to predict preferences, motivation, pleasure, learning or human errors—these are the concerns of psychological theories. On the other hand, we can say that some things will be impossible and some things will be difficult; moreover, with algebra it is routine to find such issues and very easy to do so at a very early stage in design. We can also determine that some things are possible, and if some of those are dangerous or costly the designer may wish to draw on psychological theory to make them less likely to occur given probable user behavior. We can make approximate predictions on timings; button matrices typically correspond one-to-one with user actions, so fewer matrices means there are quicker solutions, and more matrices mean user actions must take longer: we expect a correlation, which could of course be improved by using a keystroke level or other timing model.

2.3 From Finite State Machines to Matrix Algebra

Finite state machines (FSMs) are a basic formalism for interactive systems. Finite state machines have had a long history in user interface design, starting with Newman [1969] and Parnas [1964] in the 1960s, and reaching a height of interest in user interface management systems (UIMS) work [Wasserman 1985]; see Dix et al. [1998] for a textbook introduction with applications of FSMs in HCI. FSMs can handle parallelism, non-determinism, and so forth (many parallel reactive programming languages compile into FSM implementations [Magee and Kramer 1999]).

Now the concerns of HCI have moved on [Myers 2002], beyond any explicit concern with FSMs (indeed, many elementary references in HCI dismiss FSMs, e.g., Dix et al. [1998])—a continual, technology-driven pressure, but one that



a. Light with push-on/push-off action.

b. Light with separate on/off actions.

Fig. 1. Transition diagrams for alternative designs of a simple two-state system. Note that the right hand diagram does not specify what repeated user actions do: illustrating how easy it is to draw diagrams that look sensible, but which have conceptual errors. If the right hand diagram was used as a specification for programmers, the system implemented from it might do unexpected and undesirable things (e.g., the diagram does not specify what doing 'off' does when the system is already off).

tends to leave open unfinished theoretical business. However, FSMs are formally isomorphic to matrix algebra, and they can be used as a familiar introduction to the matrix algebra approach this article takes. (The Appendix to this article formalizes the isomorphism and discusses non-deterministic FSMs, which, however, are not used in the body of this article. The formal basis is not required to follow the article.)

FSMs are often drawn as transition diagrams. A transition diagram consists of circles and arrows connecting the circles; the circles represent states, and the arrows represent transitions between states. Typically both the circles and the arrows are labelled with names. A finite state machine is in one state at a time, represented by being 'in' one circle. When an action occurs, the corresponding arrow from that state is followed, taking the machine to its next state. Figure 1 illustrates two very simple transition diagrams, showing alternative designs of a simple system.

A FSM labels transitions from a finite set. Labels might be button names, ON, OFF, REWIND, say, corresponding to button names in the user interface. In our matrix representation, each transition label denotes a matrix, B_1, B_2, B_3 ..., or in general B_i. Buttons and matrices are not the same thing: one is a physical feature of a device, or possibly the user's conception of the effect of the action; the other is a mathematical object. Nevertheless, except where confusion might arise in this article, it is convenient to use button names for matrices interchangeably. In particular this saves us inventing mathematical names for symbolic buttons, such as $\boxed{\vee}$. (See the Appendix for a formal perspective on this convenience.)

The state of the FSM can be represented by a vector, $\mathbf{s}$. When a transition occurs, the FSM goes into a new state. If the transition is represented by the matrix B_i, the new state is $\mathbf{s}$ times B_i, written $\mathbf{s}B_i$. Thus finding the next state amounts to a matrix multiplication.

If we consider states to be labelled 1 to N, then a convenient representation of states is by unit vectors, e_s, a vector of 0s of length N, with a 1 at the position corresponding to the state number s; under this representation, the matrices B will be $N \times N$ matrices of 0s and 1s (and with certain further interesting properties we do not need to explore here). Now the matrix multiplication $e_s B_i =$

e_t means "doing action i in state s puts the system in state t," and several multiplications such as $e_s B_i B_j = e_t$ means "doing action i then action j starting from state s puts the system in state t."

Instead of having to draw diagrams to reason about FSMs, we now do matrix algebra. However big a FSM is, the formulas representing it are the same size: "$\mathbf{s}B_1B_2$" could equally represent the state after two transitions on a small 4 state FSM or on a large 10 000 state FSM. The size of the FSM and its details are completely hidden by the algebra. Moreover, since any matrix multiplication such as B_1B_2 gives us another matrix, a single matrix, say $M = B_1B_2$, can represent any number of user actions: "$\mathbf{s}M$" might represent the state after two button presses, or more.

Matrix algebra can represent task/action mappings too; suppose, as a simple case, that a user wants to get to some state t from an initial state s. How can they do this? However as ordinary users they go about working out how to solve their task, or even using trial and error, their thinking is equivalent to solving the algebraic problem of finding a matrix M (possibly not the entire matrix) such that $sM = t$, and then finding a factorization of M as a product of matrices B_1, B_2, and so on, that are available as actions to them, such as $M = B_1B_2$, meaning in this case that two actions are sufficient: $sB_1B_2 = t$.

Putting the user's task into matrices may make it sound more complicated than one might like to believe, but all it is doing is spelling out exactly what is happening. Besides, most user interfaces are not easy to use, and the superficial simplicity of ignoring details is deceptive.

3. INTRODUCTORY EXAMPLES

This section provides some simple motivating examples.

3.1 A Simple Two Button, Two State System

Imagine a light bulb controlled by a pushbutton switch. Pressing the switch alternately turns the light on or off. This is a really simple system, but sufficient to clearly illustrate how button matrices work. Figure 1a presents a state transition diagram for this system.

There are two states for this system: the bulb is on or off. We can represent the states as a vector, with on as $(1\ 0)$ and off as $(0\ 1)$. The pushbutton action push is defined by a 2×2 matrix:

$$\boxed{\texttt{PUSH}} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Such simple 2×2 matrices hardly look worth using in practice. Writing them down seems as hard as informally examining what are obvious issues. In fact, all elementary introductions to matrices have the same problem.[3] The point

[3]You might first learn how to do two variable simultaneous equations, but next learning how to use 2×2 matrices to solve them further requires learning matrix multiplication, inverses and so on, and the effort does not seem to be adequately rewarded, since you could more easily solve the equation without matrices! However if you ever came across four, five or more variable equations—which you rarely do in introductory work—the advantages become stark.

is that the matrix principles, while very obvious with such a simple system, also apply to arbitrarily complex systems, where thinking systematically about interaction would be impractical. With a complex system, the matrix calculations would be done by a program or some other design or analysis tool: the designer or user would not need to see any details. For complex systems, the user interface properties are unlikely to be obvious except by doing the matrix calculations. For large, complex systems, matrices promise rigour and clarity without overwhelming detail.

Doing the matrix multiplication we can check that if the light is off then pushing the button makes the light go on:

$$\begin{aligned} \textbf{off}\,\boxed{\texttt{PUSH}} &= (0\ 1)\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \\ &= (1\ 0) \\ &= \textbf{on} \end{aligned}$$

Similarly, we can show that pushing the button when the light is on puts it off, since $\textbf{on}\,\boxed{\texttt{PUSH}} = \textbf{off}$.

It seems that pushing the button twice does nothing, as certainly when the light is off, one push puts it on, and a second puts it off. We could also check that pressing $\boxed{\texttt{PUSH}}$ when it is on puts it off, and pressing it again puts it on, so we are back where we started. Rather than do these detailed calculations, which in general could be very tedious as we would normally expect to have far more than just two states, we can find out what a double press of push means *in any state*:

$$\begin{aligned} \boxed{\texttt{PUSH}}\,\boxed{\texttt{PUSH}} &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \\ &= I \end{aligned}$$

Thus the matrix multiplication $\boxed{\texttt{PUSH}}$ times $\boxed{\texttt{PUSH}}$ is equal to the identity matrix I. We can write this in either of the following ways:

$$\begin{aligned} \boxed{\texttt{PUSH}}\,\boxed{\texttt{PUSH}} &= I \\ \boxed{\texttt{PUSH}}^2 &= I \end{aligned}$$

Anything multiplied by I is unchanged—a basic law of matrices. In other words, doing $\boxed{\texttt{PUSH}}$ then $\boxed{\texttt{PUSH}}$ does nothing. Thus we now know without doing any calculations on each state, that $\boxed{\texttt{PUSH}}\,\boxed{\texttt{PUSH}}$ leaves the system in the same state, for every starting state.

From our analysis so far, we have established that a user can change the state of the system or leave it unchanged, even if they changed it. This is a special case of undo: if a user switched the light on we know that a second push would switch it off and *vice versa*.

In general a user may not know what state a system is in, so to do some useful things with it, user actions must have predictable effects. For example,

if a lamp has failed (so we do not know whether the electricity is on or off), we must switch the system off to ensure we can replace the lamp safely, to avoid any risk of electrocution.

Let us examine this task in detail now. A quite general state of the system is **s**. The only operation the user can do is press the pushbutton, but they can choose how may times to press it; in general the user may press the button n times—that's all they can do. It would be nice if we could find an n and then tell the user that if they want to do a "replace bulb safely" task, they should just press PUSH n times. If n turns out to be big, it would be advisable for the design to be such that the solution to the task be "press PUSH at least n times" (in other words, so larger n is also acceptable) in case the user misses one or loses count.

We will see below (Section 4) that for a Nokia mobile phone, the answer to a similar question is $n = 4$. Nokia, noticing that, provided a feature so that holding the button down continuously is equivalent to pressing it four times, thus providing a simpler way for a user to solve the task.

Back to the light system. If we want to know how to get the light off we need to find an n (in fact an $n \geq 0$) such that

$$\boxed{\texttt{PUSH}}^n = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}$$

since this matrix is the only matrix that guarantees the light will be off whatever state it operates on. Note that the user interface does not provide the matrix directly; instead we are trying to find some sequence of available operations that combine to be equal to the matrix describing the user's task. For this particular system, the only sequences of user operations are pressing PUSH, and these sequences of operations can only differ in their length. Thus we have a simple problem, merely to find n rather than a complicated sequence of matrices. It is not difficult to prove rigorously that there is no solution to this equation.

For this system, the user cannot put it in the off state (we could also show it is impossible to put it in any particular state) without knowing what state it is in to start with. Clearly we have a system design inappropriate for an application where safe replacement of failed lamps is an issue. If a design requirement was that a user should be able to do this, then the user interface must be changed, for instance to a two-position switch.

A two-position switch gives the user two actions: ON and OFF.

$$\boxed{\texttt{ON}} = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}$$

$$\boxed{\texttt{OFF}} = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}$$

We can check that OFF works as expected whatever state the system is in:

$$\mathbf{off}\,\boxed{\texttt{OFF}} = (0\ 1)\begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}$$

$$
\begin{aligned}
&= (0\ 1) \\
&= \textbf{off} \\
\textbf{on}\ \boxed{\texttt{OFF}} &= (1\ 0)\begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \\
&= (0\ 1) \\
&= \textbf{off}
\end{aligned}
$$

Thus, regardless of the initial state, pressing $\boxed{\texttt{OFF}}$ results in the lamp being off (we now have a solution for the task). The explicit check, above, that this is so seemed tedious because it involved as many matrix calculations as there are states. Could we reason better from a diagram? Unfortunately, simple as Figure 1b seems, it presents a state transition diagram that only resembles this system (the diagram omits some transitions that are defined in the two matrices) and we cannot reason reliably from it. We cannot even tell, just by looking at the diagram, that it is faulty! For larger systems, we certainly need a more reliable approach than informal reasoning based on pictures, and certainly we will need techniques that are effective when there are many more, even millions, of states (and diagrams for such systems would be so complex that they would be useless). In general, a better approach must be more abstract.

Here there are just two user actions (and in general there will be many fewer actions than there are states): we have just $\boxed{\texttt{ON}}$ and $\boxed{\texttt{OFF}}$. Let us consider the user doing anything then $\boxed{\texttt{ON}}$. There are only two possibilities: the user's action can follow either an earlier $\boxed{\texttt{ON}}$ or an earlier $\boxed{\texttt{OFF}}$:

$$
\begin{aligned}
\boxed{\texttt{ON}}\,\boxed{\texttt{ON}} &= \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \\
&= \boxed{\texttt{ON}} \\
\boxed{\texttt{OFF}}\,\boxed{\texttt{ON}} &= \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \\
&= \boxed{\texttt{ON}}
\end{aligned}
$$

These are calculations purely on the matrices, not on the states. (Coincidentally, and somewhat misleadingly, for this simple system there happen to be as many user actions as there are states, and we ended up calling them with the same names to confound the confusion!) The point is that we are now using the algebra to deal with things the user does—actions—and this is generally much easier to do than to look at the states.

In the two cases above the result is equivalent to pressing a single $\boxed{\texttt{ON}}$. We could do the same calculations where the second action is $\boxed{\texttt{OFF}}$, and we would find that if there are two actions and the second is $\boxed{\texttt{OFF}}$ the effect is the same as a single $\boxed{\texttt{OFF}}$. This system is *closed*, meaning that any combination of actions is equivalent to a single user action. Closure is an important user interface

property (it may not be relevant for some designs or tasks): it guarantees anything the user can do can always be done by a single action.

This system is not only closed, but furthermore any sequence of actions is equivalent to the last action the user does. Here is a sketch of the proof. Consider an arbitrarily long sequence S of user actions $A_1A_2 \ldots A_n$ for this system. ($S = A_1A_2 \ldots A_n$ is just a matrix multiplication.) We have calculated that the system is closed for any two operations: thus (as we confirmed above by explicit calculation) A_1A_2 must be equivalent to either $\boxed{\texttt{ON}}$ or $\boxed{\texttt{OFF}}$, and in fact it is equal to A_2. But this followed by A_3 will be either $\boxed{\texttt{ON}}$ or $\boxed{\texttt{OFF}}$, so *that* followed by A_4 will be too ... it's not hard to follow the argument through and conclude that $S = A_n$. Put in other words, after any sequence of user actions, the state of the system is fully determined by the last user action. Specifically if the last thing the user does is switch off, *whatever* happened earlier, the system will be off; and if the last thing the user does is switch on, *whatever* happened earlier, the system will be on. We now have a system that makes solving the task of switching off safely and reliably even when not knowing the state (or any of the previous history of use of the system) very easy. For the scenario we were imagining, this design will be an easier and a much safer system.

None of this is particularly surprising, because we are very familiar with interactive systems that behave like this. And the two designs we considered were very simple systems. What, then, have we achieved? I showed that various system designs have usability properties that can be expressed and explored in matrix algebra. I showed we can do explicit calculations, and that the calculations give us what we expect, although more rigorously. We can do algebraic reasoning, in a way that does not need to know or depend on the number of states involved.

3.2 Simple Abstract Matrix Examples

Having seen what matrices can do for small concrete systems, we now explore some usability properties of systems in general—where, because of their complexity, we typically do not know beforehand what to expect.

Matrix multiplication does not commute: if A and B are two matrices, the two products AB and BA are generally different. This means that pressing button A then B is generally different from pressing B then A. The last system was non-commutative, since for it $\boxed{\texttt{ON}}\,\boxed{\texttt{OFF}} \neq \boxed{\texttt{OFF}}\,\boxed{\texttt{ON}}$. This is not a deep insight, but it is a short step from this sort of reasoning to understanding undo and error recovery, as we shall see below.

In a direct manipulation interface, a user might click on this or click on that in either order. It is important that the end result is the same in either case. Or in a pushbutton user interface there might be an array of buttons, which the user should be able to press in any order that they choose. Both cases are examples of systems where we *do* want the corresponding matrices to commute. We should therefore either check $\boxed{\texttt{Click}}_1\,\boxed{\texttt{Click}}_2 = \boxed{\texttt{Click}}_2\,\boxed{\texttt{Click}}_1$ by proof or direct calculation with matrices, or we should design the interface to ensure the matrices have the right form to commute. Just as allowing a user to do operations in any order makes the interface easier to use [Thimbleby 2001],

the analysis of user interface design in this case becomes much easier since commutativity permits mathematical simplifications.

Suppose we want a design where pressing the button [OFF] is a shortcut for the two presses [STOP] [OFF], for instance as might be relevant to the operation of a DVD player. The DVD might have two basic modes: playing and stopped. If you stop the DVD then switch it off, this is the same as just switching it off—where it is also stopped. What can we deduce? Let S and O be the corresponding matrices; in principle we could ask the DVD manufacturer for them. The simple calculation $SO = O$ will check the claim, and it checks it under all possible circumstances—the matrices O and S contain *all* the information for all possible states. This simple equation puts some constraints on what S and O may be. For instance, assuming S is non trivial, we conclude that O is not invertible. We prove this by contradiction.

Assume $SO = O$, and assume O is invertible. If so, there is a matrix O^{-1} that is the inverse of O. Follow both sides by this inverse: $SOO^{-1} = OO^{-1}$ which can be simplified to $SI = I$, as $OO^{-1} = I$. Since $SI = S$ we conclude that $S = I$. Hence S is the identity matrix, and [STOP] does nothing. This is a contradiction, and we conclude that if O is a short cut then it cannot be invertible. If it is not invertible, then in general a user will not be able to undo the effect of [OFF].

What not being invertible means, more precisely, is that the user cannot return to a previous state only knowing what they have just done. They also need to know what state the device was in before the operation and be able to solve the problem of pressing the right buttons to reach that state.[4]

To summarize, if we had the three seemingly innocuous design requirements:

(1) [STOP] does something (such as switching the playing off!)
(2) [OFF] is a shortcut for [STOP] [OFF]
(3) [OFF] is undoable or invertible (e.g., so [ON], would get the DVD back to whatever mode it was in before switching off—that is, where [ON] = [OFF]$^{-1}$).

Then we have just proved that they are inconsistent—if we insist on them, we end up building a DVD player that *must* have bugs and *must* have a user manual that is misleading too. Better, to avoid inconsistency, the designer must forego one or more of the requirements (here, the third requirement is obviously too strict), or the designer can relax the requirements from being universal (fully true over all states) to partial requirements (required only in certain states). I discuss partial theorems below, in Section 4.2.

We now turn from these illustrative examples to some real case studies.

4. EXAMPLE 1: THE NOKIA 5110 MOBILE PHONE

The menu system of the Nokia 5110 mobile phone can be represented as a FSM of 188 states, with buttons [∧], [∨], [C], and [NAVI] (the Nokia context sensitive button: the meaning is changed according to the small screen display). In this

[4]Or the user needs to know an algorithm to find an undo: for instance, to be able to recognize the previous state, and be able to enumerate every state, would be sufficient—but hardly reasonable except on trivial devices.

ACKNOWLEDGMENTS

Harold Thimbleby is a Royal Society-Wolfson Research Merit Award Holder, and acknowledges their support. The author is also grateful for very constructive comments from David Bainbridge, Ann Blandford, George Buchanan, Paul Cairns, George Furnas, Jeremy Gow (who suggested partial theorems), Michael Harrison, Mark Herbster, Tony Hoare, Matt Jones and Peter Pirolli.

REFERENCES

ADDISON, M. A. AND THIMBLEBY, H. 1996. Intelligent adaptive assistance and its automatic generation, *Interact. Comput. 8*, 1, 51–68.

ALTY, J. L. 1984. The application of path algebras to interactive dialogue design. *Behav. Infor. Tech.*, *3*, 2, 119–132.

ANDERSON, J. R. AND LEBIERE, C. 1998. *The Atomic Components of Thought*, Lawrence Erlbaum Associates.

BLANDFORD, A. AND YOUNG, R. M. 1996. Specifying user knowledge for the design of interactive systems, *Soft. Eng. J. 11*, 6, 323–333.

BROWN, J. R. 1999. *Philosophy of Mathematics*, Routledge.

BROYDEN, C. G. 1975. *Basic Matrices*, Macmillan.

CARD, S. K., MORAN, T. P., AND NEWELL, A. 1983. *The Psychology of Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ.

CLARKE, Jr., E. M., GRUMBERG, O., AND PELED, D. A., 1999. *Model Checking*, MIT Press.

CURZON, P. AND BLANDFORD, A. 2001. Detecting Multiple Classes of User Errors. M. Reed Little & L. Nigay, eds. *Engineering for Human-Computer Interaction*, Springer Verlag, LNCS *2254*, 57–71.

DIX, A., FINLAY, J., ABOWD, G., AND BEALE, R. 1998. *Human-Computer Interaction*, 2nd. ed., Prentice Hall.

GATHEN, J. VON ZUR AND GERHARD, J. 2003. *Modern Computer Algebra*, Cambridge University Press.

GOW, J. AND THIMBLEBY, H. 2004. MAUI: An interface design tool based on matrix algebra. *ACM Conference on Computer Aided Design of User Interfaces*, CADUI IV, R. J. K. Jacob, Q. Limbourg & J. Vanderdonckt, Eds. 81–94 (pre-proceedings page numbers).

HARTSON, H. R., SIOCHI, A., AND HIX, D. 1990. The UAN: A user oriented representation for direct manipulation interface designs. *ACM Trans. Infor. Syst. 8*, 3, 181–203.

HORROCKS, I. 1999. *Constructing the User Interface with Statecharts*, Addison-Wesley,

IVORY, M. Y. AND HEARST, M. A. 2001. The state of the art in automating usability evaluation of user interfaces. *ACM Comput. Surv. 33*, 4, 470–516.

JOHNSON, S. C. 1979. Yacc: Yet another compiler-compiler. *UNIX Programmer's Manual*, *2*, Holt, Rinehart, and Winston, New York, NY, 353–387.

KIERAS, D. AND POLSON, P. G. 1985. An approach to the formal analysis of user complexity. *Int. J. Man-Mach. Studies 22*, 4, 365–394.

KNUTH, D. E. 1992. Two notes on notation. *Am. Math. Monthly*, *99*, 5, 403–422.

LAMPORT, L. 1995. TLA in pictures. *IEEE Trans. Soft. Eng. 21*, 9, 768–775.

LIEBLER, R. A. 2003. *Basic Matrix Algebra with Algorithms and Applications*, Chapman & Hall/CRC.

MAGEE, J. AND KRAMER, J. 1999. *Concurrency: State Models & Java Programs*, John Wiley and Sons.

MYERS, B. 2002. Past, present, and future of user interface software tools. J. M. Carroll. In *Human-Computer Interaction in the New Millenium*, Ed. Addison-Wesley.

NEWMAN, W. M. 1969. A system for interactive graphical programming. In *Proceedings 1968 Spring Joint Computer Conference*. American Federation of Information Processing Societies, 47–54.

PARNAS, D. L. 1964. On the use of transition diagrams in the design of a user interface for an interactive computer system. In *Proceedings of the 24th. ACM National Conference*, 379–385.

PAYNE, S. J. 1991. Display-based action at the user interface, *Int. J. Man-Mach. Studies 35*, 3, 275–289.

PIROLLI, P. AND CARD, S. K. 1998. Information foraging models of browsers for very large document spaces. *ACM Proceedings Advanced Visual Interfaces, AVI'98*, 83–93.

RASKIN, J. 2000. *The Humane Interface*, Addison-Wesley.

SMITH, D. C. S., IRBY, C., KIMBALL, R., VERPLANK, B., AND HARLEM, E. 1982. Designing the star user interface. *Byte 7*, 4, 242–282.

THIMBLEBY, H. 1990. *User Interface Design*, ACM Press Frontier Series, Addison-Wesley.

THIMBLEBY, H. 1999. Specification-led design for interface simulation, collecting use-data, interactive help, writing manuals, analysis, comparing alternative designs, etc. *Pers. Tech. 4*, 2, 241–254.

THIMBLEBY, H. 2000. Calculators are needlessly bad. *Int. J. Human-Comput. Studies 52*, 6, 1031–1069.

THIMBLEBY, H., CAIRNS, P., AND JONES, M. 2001. Usability analysis with markov models. *ACM Trans. Comput. Human Inter. 8*, 2, 99–132.

THIMBLEBY, H. 2000. Analysis and simulation of user interfaces. *Human Computer Interaction 2000*, S. McDonald, Y. Waern and G. Cockton, Eds. BCS Conference on Human-Computer Interaction, *XIV*, 221–237.

THIMBLEBY, H. 2001. Permissive user interfaces, *Int. J. Human-Comput. Studies 54*, 3, 333–350.

THIMBLEBY, H. 2002. Reflections on symmetry. Proceedings on *Advanced Visual Interfaces*, AVI2002, 28–33.

THIMBLEBY, H. AND GOW, J. 2004. Computer algebra in interface design research, *ACM/SIGCHI International Conference on Intelligent User Interfaces*, IUI 04, N. J. Nunes & C. Rich, Eds. 366–367.

WASSERMAN, A. I. 1985. Extending state transition diagrams for the specification of human computer interaction. *IEEE Trans. Soft. Eng. SE-11*, 8, 699–713.

WOLFRAM, S. 1999. *The Mathematica Book*, 4th. ed., Cambridge University Press.

YOUNG, R. M. 1983. Surrogates and Mappings: Two Kinds of Conceptual Models for Interactive, *Mental models*, Devices, D. Gentner and A. L. Stevens. Hillsdale, NJ: Lawrence Erlbaum Assoc., 35–52.

YOUNG, R. M., GREEN, T. R. G., AND SIMON, T. 1989. Programmable user models for predictive evaluation of interface designs. *ACM Proceedings CHI'89*, 15–19.

Received September 2002; revised July 2003, February 2004; accepted March 2004

Accepted by George Furnas