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 paper 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 paper 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 (D.2.2, H.1.2, I.3.6)—*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 generalises beyond specific case studies. This paper addresses these problems by showing how matrix algebra can be applied to user interface design. The paper explains the theory in detail and shows it applied to three real case studies.

Address: UCLIC, University College London, Remax House, 31/32 Alfred Place, LONDON, WC1E 7DP, UK. URL: `http://www.uclic.ucl.ac.uk`

1 Introduction **1**
2 Contributions to HCI **3**
2.1 Methodological issues . . . 4
2.2 The theory proposed . . . 6
3 Introductory examples **7**
3.1 A simple two button, two state system . . . 7
3.2 Simple abstract matrix examples . . . 11
4 Example 1: The Nokia 5110 mobile phone **12**
4.1 Inverses . . . 14
4.2 Partial theorems . . . 15
5 Projecting large state spaces onto smaller spaces **15**
5.1 Matrix transforms . . . 16
6 Example 2: The Casio HL-820LC calculator **19**
6.1 Solving user problems: Task/action mappings . . . 24
6.2 Summary and comparisons with other designs . . . 30
7 Example 3: The Fluke 185 digital multimeter **31**
7.1 Specifying the Fluke 185 . . . 32
7.2 Qualifications . . . 35
7.3 Reordering states . . . 36
7.4 Remaining major features . . . 37
7.5 Soft buttons . . . 38
8 Advanced issues and further work **39**
8.1 Button modes . . . 39
8.2 Modes and partial theorems . . . 42
8.3 Finding matrices from specifications . . . 43
8.4 Design tools . . . 45
8.5 Constructive use of matrices . . . 46
8.6 Model checking and other approaches . . . 46
8.7 Modes and the ordering states . . . 47
8.8 Complexity theory . . . 47
8.9 Pseudo inverses . . . 47
9 Conclusions **48**
A Simple matrix algebra **51**
A.1 Partitions . . . 53
A.2 Algebraic properties . . . 54
B From finite state machines to matrix algebra **54**
B.1 Aren't FSMs too restricted? . . . 55
C Formal definitions and basic properties **57**

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 paper 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. Secondly, 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, and so on, in very general ways. This paper 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 synchronise. Thus a button 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 represent the user thinking about how things happen. The algebra does not 'look' towards the system nor towards the user. As used in this paper, 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 Appendix A. There are many textbooks available on matrix algebra (linear algebra); Broyden's *Basic Matrices* [6] is one that emphasises partitions, a technique that is used extensively later in this paper. Readers who want a formal background should refer to Appendix C.

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 [3], that are psychologically sophisticated models. When suitably set up, these models can make predictions about user performance and behaviour. 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 assump-

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

tions. 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 [17], UAN [13], GOMS [7], PUM [36], IL [4] and so on (see [15] 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 behaviour. Some HCI theories, such as information foraging [24], have a weaker base in psychology, but their aim, too, is to make predictions of user behaviour. 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 paper 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 §8.3). 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 paper makes is that it shows how interaction with real systems can be analysed 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. I emphasise that throughout this paper we see 'inside' matrices. In many ways, the contents of a matrix and how one calculates with it can normally be handed over to programs; the inside details are irrelvant to designers. For this paper, however, it is important to see that the approach works and that the calculations are valid. A danger is that this paper gives a misleading impression of complexity, whereas it is intended to give an accurate impression how the approach works, and how from a mathematical point of view it is 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 paper makes a theoretical contribution to HCI. One might consider that there are two broad areas of theoretical contributions, which aspire, respectively, to psychological or computational validity. This paper 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 Appendix C). 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 paper provides a whole variety of non-trivial design insights, both general approaches and 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 paper necessarily covers simple examples, but somehow I must imply bigger issues can be addressed. My approach has been to start with three real devices. Being real devices, any reader of this paper 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 specifications — I had to reconstruct them. Whilst 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 paper 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 be very hard to tell the difference between what I had done and what I should have done.

For this paper, I have also chosen relatively cheap handheld devices. Handheld devices are typically used for specific tasks. Again, this makes both replication of

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

this work and its exposition easier. However, further work might wish to consider similar sorts of user interface 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.

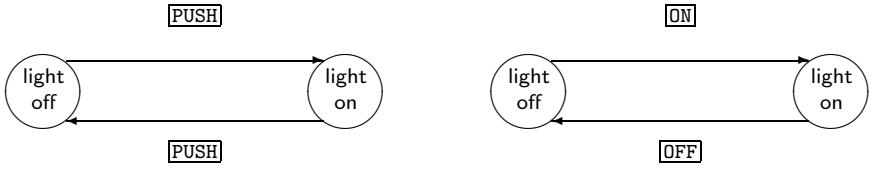
2.2 The theory proposed

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. 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 [5] 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. Factorisation 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 factorisation, it will not be easy. Indeed, we know that users keep track of very little in their heads [23], so users are like 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 paper is never going to predict preferences, motivation, pleasure, learning or probable errors — these are the concerns of psychological theories. On the other hand, we can say that some things will be impossible and some



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).

things will be difficult; moreover, with algebra it is routine to find such issues and 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 behaviour. 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.

3. INTRODUCTORY EXAMPLES

This section provides some simple motivating examples, though it avoids dwelling on where matrices 'come from.' Appendix B follows an alternative expository approach: matrices are defined by starting with finite state machines; Appendix C defines a formal correspondence between finite state machines and matrices.

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:

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

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 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 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 push then 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.

[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.

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 electrocution.

Let us examine this task in detail now. A quite general state of the system is $\mathbf{s}$. 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 solution to the task to be "press PUSH at least n times" in case the user misses one or loses count.

We will see below (§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 only differ in their length. Thus we have a simple problem, meremly 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. 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:

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

Figure 1b presents a state transition diagram that merely resembles this system: the diagram omits some obvious (or maybe not so obvious) transitions that are

defined in the two matrices. We can check that ON works as expected whatever state the system is in:

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

This check involves as many matrix calculations as there are states. This simple system only has two states, so it isn't onerous to study, but we really need techniques that are easy to use when there are even millions of states. In general, a better approach will 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 ON and OFF. Let us consider the user doing anything then ON. There are only two possibilities: the user's action can follow either an earlier ON or an earlier 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 too 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 ON. We could do the same calculations where the second action is OFF, and we would find that if there are two actions and the second is OFF the effect is the same as a single 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 sequence S of user actions $A_1 A_2 \dots A_n$ for this system. ($S = A_1 A_2 \dots A_n$ is just a matrix multiplication.) We have calculated that the system is closed for any two operations: $A_1 A_2$ must be equivalent to either [ON] or [OFF], and in fact it will be equal to A_2. But this followed by A_3 will be either [ON] or [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 reliably when not knowing the state very easy. For the scenario we were imagining, this design will be 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 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 [ON] [OFF] $\neq$ [OFF] [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 $[\texttt{Click}]_1 [\texttt{Click}]_2 = [\texttt{Click}]_2 [\texttt{Click}]_1$ 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 [30], 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} which 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 summarise, if we had the three seemingly innocuous design requirements:

(1) STOP does something (such as switching the DVD 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 second 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 §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 paper I use a definition of the Nokia 5110, as published in full elsewhere [29].

First, we describe the user interface informally in English. The menu structure is entered by pressing NAVI and then using ∧ and ∨ to move up and down the menu. Items in the menu can be selected by pressing NAVI, and this will either access a phone function or select a submenu. The submenu, in turn, can be navigated up and down using ∧ and ∨, and items within it selected by NAVI. The C key is used for correction, and 'goes up a level' to whatever menu item was selected before the last NAVI press. If the last press of NAVI selected a phone function, then C cannot correct it — once a function is selected, the phone does the function and

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

then reverts to the last menu position. The phone starts in a standby state, and in this state C does nothing.

We may hope or expect the Nokia's user interface to have certain properties. It may have been designed with certain properties in mind. Perhaps Nokia used a system development process that ensured the phone had the intended properties. Be all this as it may, I will now show that from a matrix definition of the Nokia we can reliably deduce and check properties.

We represent the buttons and button presses by boxed icons like ∧ and C, and we would normally represent the matrices they represent by mathematical names like U and C, which for the present model of the Nokia are in fact 188×188 matrices. But the buttons and matrices correspond, and they are essentially the same thing: we may as well call the mathematical objects by names which are the button symbols themselves. So although our calculations look like sequences of button presses, they are matrix algebra.

We can establish, amongst others, the following laws:

$$\begin{aligned} \boxed{\wedge}\boxed{\vee} &= I \\ \boxed{\vee}\boxed{\wedge} &= I \\ \boxed{\wedge}\boxed{\mathtt{C}} &= \boxed{\mathtt{C}} \\ \boxed{\vee}\boxed{\mathtt{C}} &= \boxed{\mathtt{C}} \\ \boxed{\mathtt{C}}^i &\neq \boxed{\mathtt{C}}^j \text{ for } 0 \leq i \leq 3,\ i \neq j \\ \text{but } \boxed{\mathtt{C}}^i &= \boxed{\mathtt{C}}^4 \text{ for } 4 \leq i \end{aligned}$$

Here, as usual I is the identity matrix. These are not just 'plausible' laws the user interface happens to obey or might often obey, or we would like it to obey: we calculated these identities: they are universally true, facts that can be established directly using the 188×188 matrices from the Nokia specification we started from (in fact, we wrote a program to look for interesting identities — I had no preconceptions on what to find).

Some of these identities are not surprising: doing up then down (or down then up — one does not imply the other) has no effect; although it might be surprising that it *never* has any effect, which is what the identity means.

Up or down followed by C is the same as if C had been pressed directly; on the other hand, NAVI C is not the same as C, since when NAVI activates a phone function the C key cannot correct it.

Finally, direct calculation shows that $\boxed{\mathtt{C}}^4 = \boxed{\mathtt{C}}^5$, and moreover that this is the least power where they are equal. If they are equal, they will be equal if we do the same things to both sides of the equation, so $\boxed{\mathtt{C}}^4\boxed{\mathtt{C}} = \boxed{\mathtt{C}}^5\boxed{\mathtt{C}}$ and hence $\boxed{\mathtt{C}}^5 = \boxed{\mathtt{C}}^6$. By induction, $\boxed{\mathtt{C}}^i = \boxed{\mathtt{C}}^{i+1}$ for all $4 \leq i$, and hence

$$\boxed{\mathtt{C}}^i = \boxed{\mathtt{C}}^4 \text{ for } 4 \leq i$$

The identity means that if C is pressed often enough, namely at least 4 times, further presses will have no effect (an idempotence law). In fact, Nokia recognise the value of this: if the C key is held down continuously for a couple of seconds, it

behaves like $\boxed{\texttt{C}}^4$ — thus the Nokia 5110 also has a user action 'hold C' with matrix defined by

$$\boxed{\texttt{hold C}} = \boxed{\texttt{C}}^4$$

4.1 Inverses

Matrices only have inverses if their determinants are non-zero. A property of determinants is that the determinant of a product is the product of the determinants: for any matrices A and B:

$$\det(AB) = \det(A)\ \det(B)$$

In a product, if any factor is zero, the entire product is zero — zero times anything is zero. So if any determinant is zero, the determinant of the entire product will be zero. What this means for the user is that when they do a sequence of button presses corresponding to the matrix product $B_1 B_2 \ldots B_n$, if *any* of them are not invertible (not undoable), the entire sequence will not be invertible. So buttons with matrices that have zero determinants (i.e., are singular) are potentially dangerous: if they are used in error, there is no uniform way to recover from the mistake. The user might have fortuitously kept a record of or memorised their actions up to the point where they made the mistake, and then they might be able to recover using the device's buttons, but if so they are also having to use this additional knowledge, something external the device cannot help with.

If a matrix cannot be inverted (because it is singular) the user cannot undo the effect of the corresponding button, but even if a matrix can be inverted in principle, in practice the user may not be able to undo its effect: they may not have access to all buttons that are factors of the inverse. The user is only provided with *particular* buttons and hence can only construct particular matrices. A routine calculation can establish what the user can do with their actual buttons; a designer may wish to check that every button's inverse is a product of at least one other button. For example, the determinants of $\boxed{\vee}$ and $\boxed{\wedge}$ on the Nokia mobile phone are both -1, which is non-zero, so these matrices can be inverted. The user might have broken the $\boxed{\vee}$ button. In this case, as $\boxed{\wedge}$ is still invertible as a matrix, but the user cannot undo its effect (at least, without knowing a lot about the Nokia and the way $\boxed{\wedge}$ works in each menu level).

In contrast to $\boxed{\vee}$ and $\boxed{\wedge}$, the matrices $\boxed{\texttt{C}}$ and $\boxed{\texttt{NAVI}}$ are both singular, which means they cannot be inverted. In other words, there is no matrix M such that $\boxed{\texttt{C}}M = I$ or $\boxed{\texttt{NAVI}}M = I$. Since there is *no* matrix, there is not even a sequence of button presses that achieves this. But if there is no matrix, there is no such product — whatever the buttons. In user interface terms, this means that if $\boxed{\texttt{C}}$ or $\boxed{\texttt{NAVI}}$ are pressed by mistake, the user cannot in general recover from the error — at least without knowing exactly what they were doing. If a matrix is not invertible, it means the device no longer knows what it was doing, and therefore it cannot go back in every case.

The strong, and in some contexts over-strict, proviso 'in every case' motivates the next section.

4.2 Partial theorems

Theorems such as $\boxed{\texttt{C}}^4 = \boxed{\texttt{C}}^5$ can be found automatically, as can the fact that $\boxed{\texttt{C}}$ is not invertible. Theorems that are true, such as these, are not the only sort of theorem that are relevant to usability.

A *partial theorem* is a theorem that is true for some states but not for all states. A user may be misled by a theorem that is, say, true in 90% of states. For example, they will use a device, and come to believe that $\boxed{\wedge}\boxed{\vee} = I$, but one day they do $\boxed{\wedge}$ but find to their surprise that $\boxed{\vee}$ does not get them back — that is, in this scenario the theorem $\boxed{\wedge}\boxed{\vee} = I$ is a partial theorem.

A designer must be interested in finding partial theorems. In particular, a designer will be interested in *simple* partial theorems — these are ones that a user can easily infer and may believe to be universal; and a designer will be especially interested in partial theorems that have safety or cost implications.

Once a partial theorem has been identified, it is then easy to find out details:

—The states in which the theorem fails can be determined. The designer can reconsider whether the device should be designed so that the theorem is true for these states. Many partial theorems may fail when a device is off; this is a case that is unlikely to raise a design issue.

—The button matrices that reveal the partiality can be determined. If a theorem is partial, it fails for some states only. The matrices that take the user to those states therefore correspond to the user actions that would reveal the failure of the theorem. If these actions are guarded (e.g., by physical covers, locks, passwords) then the user will be less likely to take action to make the theorem fail. For example, we might decide to place a flap over the $\boxed{\texttt{OFF}}$ button so it cannot be pressed by accident.

The discovery of theorems and partial theorems can be fully automated; we have used *Mathematica* as well as our own design tool [12], which (unlike *Mathematica*) hides the mathematics from designers. A brief comparison of our tool and *Mathematica* can be found in [32].

We return to partial theorems in §8.2, after introducing more examples.

5. PROJECTING LARGE STATE SPACES ONTO SMALLER SPACES

Although it is possible to use matrices to model entire systems, often it is undesirable to do so.

We may want to treat classes of states as equivalent. In a CD player, for instance, we may not really be interested in how much of which track is playing — we might just be interested in how playing works for any track. Or we may want to reason closely about a few buttons, and ignore the rest of the system. In fact we did this with the Nokia example in the last section: the Nokia mobile phone had a model of 188 states, and while this completely described the menu subsystem of the Nokia phone it did not cover any other features, such as dialling or SMS (short message service, for sending text messages). This was a pragmatic decision, and one that can be justified because other buttons on the phone (such as the digit keys) are 'obviously' irrelevant to how the menu system works. But are they really?

We need a systematic and reliable approach to getting at the pertinent properties of systems we are interested in. In this paper, I discuss two approaches to this

problem: matrices themselves can be used (and this technique will be used later in this paper), discussed below in §5.1, and computer algebra systems can transform functional specifications into matrices, discussed in §8.3.

5.1 Matrix transforms

This section shows how matrices can be used to reliably abstract out just the features that are needed.

To project a large state vector to a smaller space, multiply by an appropriate projection matrix. Simply, if the large state space has M states, the projection matrix P has M rows and N columns, then the projected state space vector $\mathbf{s}P$ will have N columns (or equivalently, N states). We then consider button matrices operating on $\mathbf{s}P$ rather than on $\mathbf{s}$: these matrices will be square $N \times N$ matrices, possibly much smaller than the original $M \times M$ size. Suppose the fully explicit button matrices are B and the projected matrices are B'. All we require is $\mathbf{s}BP = \mathbf{s}PB'$ (i.e., that $BP = PB'$) to associate with any button matrix (or button matrix product) B the smaller projected matrix B'.

For concreteness consider a digital clock, and we will be interested in the behaviour of the tens of hours setting button, TENS, and the on/off arrangements. Such clocks must display 24 hours and 60 minutes; they therefore need $24 \times 60 = 1440$ states just to display the time. We also need an extra state for off, when the clock is in a state displaying no time at all. The state occupancy vector $\mathbf{s}$ is therefore a vector with 1441 elements, which is too big to write down explicitly. Our clock has four buttons to increment the corresponding digits, so that a user can set the time. These buttons could be represented fully as 1441×1441 matrices.

We define a matrix P that projects the state space onto a smaller space, the space we are interested in exploring in detail. Suppose we want to work in the tens of hours space, in which case P will project 1441 states to 4 states: off, or displaying 0, 1, or 2 in the tens of hours column. Thus P will be a matrix with 4 columns and 1441 rows.

There is not space here to show P explicitly because it has 5764 elements, and in any case that number of elements would be hard to interpret. The definition of P depends on how states are arranged. We have to choose some convention for the projected state space; for the sake of argument take

$$\mathbf{s}P = (\text{[off?]}\ \ \text{[displaying 0?]}\ \ \text{[displaying 1?]}\ \ \text{[displaying 2?]})$$

where $[e]$ means 0 or 1 depending on whether e is true — a convenient notation due to Knuth [18]. If we assume the state vector $\mathbf{s}$ is arranged in the obvious way that state 1 is off, state 2 is displaying time 0000, state 3 is time 0001, state 4 is time 0002 … state 60 is time 0059, state 61 is time 0100 … state 1441 is time 2359, then P will look like this:

$$P = \begin{pmatrix} 1&0&0&0 \\ 0&1&0&0 \\ \vdots&\vdots&\vdots&\vdots \\ 0&1&0&0 \\ 0&0&1&0 \\ \vdots&\vdots&\vdots&\vdots \\ 0&0&1&0 \\ 0&0&0&1 \\ \vdots&\vdots&\vdots&\vdots \\ 0&0&0&1 \end{pmatrix} \begin{matrix} \text{one row, for off} \\ \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} \text{600 rows the same, for displaying 0} \\ \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} \text{600 rows the same, for displaying 1} \\ \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} \text{240 rows the same, for displaying 2} \end{matrix}$$

A possible definition of the tens of hours button matrix[5] is this:

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

The $\boxed{\texttt{TENS}}$ button leaves the clock off if it is already off, and otherwise increments the digit displayed by the tens of hours digit. We can confirm this by explicitly working out what $\boxed{\texttt{TENS}}$ does in each of the four states:

$$(1\ 0\ 0\ 0)\boxed{\texttt{TENS}} = (1\ 0\ 0\ 0)$$

$$(0\ 1\ 0\ 0)\boxed{\texttt{TENS}} = (0\ 0\ 1\ 0)$$

$$(0\ 0\ 1\ 0)\boxed{\texttt{TENS}} = (0\ 0\ 0\ 1)$$

$$(0\ 0\ 0\ 1)\boxed{\texttt{TENS}} = (0\ 1\ 0\ 0)$$

If we are only interested in the on/off switch, we do not even care what digit is displayed, and we can project the clock's 1441 states on to just two, on and off. A new projection matrix Q with 1441 rows and 2 columns is required, but it is clearer and easier to define Q in terms of P, rather than write it explicitly — here we see another advantage of projecting a huge state space onto something more manageable.

$$Q = P \begin{pmatrix} 1&0 \\ 0&1 \\ 0&1 \\ 0&1 \end{pmatrix}$$

The clock might have an on-off button:

[5] To avoid typographical clutter I shall write $\boxed{\texttt{TENS}}$ rather than $\boxed{\texttt{TENS}}'$.

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

The other buttons on this clock do not change the on/off state of the clock, so in this state space they are identities, e.g.,

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

Or perhaps the clock has two separate buttons, one for on, one for off?

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

This looks pretty simple, but we can already use these matrices to make some user interface design decisions. Suppose for technical reasons, when the clock is switched off the digits stay illuminated for a moment (this is a common design problem: due to internal capacitance the internal power supply keeps displays alight for a moment after being switched off). Users might therefore be tempted to switch the clock off again, assuming that their first attempt failed (perhaps because the switches are nasty rubber buttons with poor feedback). It is easy to see from the matrices that a repeated (specifically, double) use of $\boxed{\texttt{ON-OFF}}$ leaves the clock state unchanged, whereas any number of pressings of $\boxed{\texttt{OFF}}$ is equivalent to a single press of $\boxed{\texttt{OFF}}$. Under these circumstances — which are typical for complex push button devices like DVD players, TVs and so on[6] — we should prefer a separate off button that, unlike the $\boxed{\texttt{ON-OFF}}$ button, cannot be used to switch the device on by accident.

The scenario does *not* require an on button; what, then, about switching on? We could arrange for all of the time-setting buttons to switch the clock on, e.g.,

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

We now have a clock with five buttons. This is the same number of buttons as one with a single $\boxed{\texttt{ON-OFF}}$ button, and therefore the same build price. Furthermore, it has the nice feature that if the user attempts to set a digit by pressing a digit button (say, $\boxed{\texttt{TENS}}$) that button *always* changes what is displayed. Pressing $\boxed{\texttt{TENS}}$ would change nothing to 0, change 0 to 1, change 1 to 2, and change 2 to 0. To define this behaviour requires the previous 4 state model:

$$\boxed{\texttt{OFF}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}, \quad \boxed{\texttt{TENS}} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}, \text{ and other buttons} \ldots$$

A similar approach could be used for TVs and DVD players of course.

[6] The JVC HR-D540EK has the additional complexity that pressing $\boxed{\texttt{OPERATE}}$ slowly (what it calls the button we call $\boxed{\texttt{ON-OFF}}$ in this paper) enters a 'child lock' mode that disables the device.

6. EXAMPLE 2: THE CASIO HL-820LC CALCULATOR

Calculators differ in details of their design, and so to be specific I base our discussion on a particular calculator, the Casio HL-820LC, which is a market-leading and popular handheld (5.5×10cm) calculator. It is a current model and can be readily obtained: the discussion below should be easy to check if required. This section closes with a brief comparison with some differently designed Casio calculators. A more general critique of calculators (and a wider range of calculators) can be found elsewhere [27].

The previous section showed how a designer can project a large system down to a manageable size. Similarly, users have models of systems that are typically much smaller than the actual implementation model of the systems. We start by drawing a simple arrow diagram representing what happens to the inside state $\mathbf{s}$ of the calculator when a button is pressed:

$$\mathbf{s} \xrightarrow{\text{do } B} \mathbf{s}B$$

The user has no reasonable way of working out this because it involves understanding the calculator's program or its specification, both of which are technical documents of no interest to calculator users; after all, the whole point of the calculator is to do the work! Instead users have some sort of perception of the device and mental model, that somehow transforms something of the internal state $\mathbf{s}$ into a mental state. We can call the user's model of the state $\mathbf{m}$, and the user's model of the button matrix $\boxed{\mathtt{B}}$. We then obtain this diagram:

$$\begin{array}{lcc} \textit{implementation}: & \mathbf{s} & \longrightarrow \quad \mathbf{s}B \\ & \downarrow & \downarrow \\ \textit{user model}: & \mathbf{m} & \longrightarrow \quad \mathbf{m}\boxed{\mathtt{B}} \end{array}$$

The vertical, down pointing, arrows represent 'perception' (by whatever mechanism the user constructs their user model). The horizontal, right pointing, arrows represent actions. The diagram is a commuting diagram, since however the arrows are followed, the end result should be the same, meeting at the bottom right. The diagram is clearer than the equivalent formula: $(\text{perception}(\mathbf{s}))\boxed{\mathtt{B}} = \text{perception}(\mathbf{s}B)$. The diagram fails to commute when the user has a faulty model of the behaviour, though the fault may lie in the design (for instance, the user cannot see enough of $\mathbf{s}$ to know what state the system is in; there may be hidden modes). Later we will see an example, when the action $\boxed{\mathtt{B}}$ is the calculator's $\boxed{\mathtt{MRC}}$ action.

For considering the display and memory of a calculator, the user's model of the state $\mathbf{m}$ need only be two numbers, which we can represent as a vector of two elements: (display memory). The matrices $\boxed{\mathtt{B}}$ will then be 2×2 matrices, that operate on these vectors. Although a user is very unlikely to think explicitly using matrices, an advantage of 2×2 matrices for this paper is we can easily show and reliably calculate what the user can (perhaps not so reliably) work out.

For the calculator and a display/memory user model the transformation itself

can be represented as a matrix. To show this, for simplicity imagine a calculator that can only handle numbers 0, 1, and 2. The matrix M that represents the user transformation of the system FSM would be something like this:

$$M = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 2 \\ 1 & 0 \\ 1 & 1 \\ 1 & 2 \\ 2 & 0 \\ 2 & 1 \\ 2 & 2 \end{pmatrix}$$

This nicely transforms a 10 state implementation into a simple 2 number model that makes more sense to the user. Here we have simply, $\mathbf{m} = \mathbf{s}M$. Note that, for illustration purposes (to show the entire system implementation need not be modelled by the user), we added a 'dummy' state 0 that the user's model does not distinguish from state 1. Perhaps this is the off state, or an error state, or something else that the user is ignoring for the purposes of understanding the display/memory issues more clearly.

In summary, for working through display/memory issues, we can represent the user's model of the state of the calculator by a row vector (display memory), which we will abbreviate to $(d\ m)$. We will take d and m to be real numbers, but of course we know that any calculator will implement them using some finite binary representation (or possibly a binary-coded decimal representation). Since we are not going to do serious arithmetic (not even division by zero) in our analysis, the issue of whether d and m are finite or not, what their precision is, how the calculator handles overflow, and so on, will not arise.

Each of the calculator's functions can be represented as a matrix multiplication, as expected. Thus the key AC, which on the Casio HL-820LC clears the display but does not change the memory corresponds to a matrix C where

$$C = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$$

.

(This is the same matrix as shown generated automatically in Appendix 8.3 from an implementation of the caclculator.)

Multiplying the calculator's state by C changes it to $(0\ m)$, as can be seen by working through the general calculation:

$$(d\ m)\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} = (0\ m)$$

This is what AC does: it sets $d = 0$ and leaves m unchanged. Curiously, then, the button called AC does not mean *All* Clear!

Many calculator users press AC repeatedly. We can see that pressing AC twice

has no further effect:

$$(d\ m)\begin{pmatrix}0&0\\0&1\end{pmatrix}\begin{pmatrix}0&0\\0&1\end{pmatrix} = (0\ m)$$

In fact, since

$$\begin{pmatrix}0&0\\0&1\end{pmatrix}^2 = \begin{pmatrix}0&0\\0&1\end{pmatrix}$$

we can prove that $\boxed{\texttt{AC}}^n = \boxed{\texttt{AC}}$ for all $n > 0$; it is clear that pressing $\boxed{\texttt{AC}}$ has no effect beyond what can be achieved by pressing it just once (in technical terms, it is idempotent). Users who repeatedly press the $\boxed{\texttt{AC}}$ button seem superstitious "we must press it once, so pressing more times will be better" — or are not certain that the $\boxed{\texttt{AC}}$ button works reliably. On the other hand, if users feel they need to press $\boxed{\texttt{AC}}$ multiple times to ensure it is definitely pressed, what does this say about the perceived reliability of other buttons needed for a calculation?

The Casio HL-820LC has other buttons. What are their corresponding 2×2 matrices? Here are some definitions:

$$\boxed{\texttt{M+}} = \begin{pmatrix}1&1\\0&1\end{pmatrix} \quad \text{add display to memory}$$

$$\boxed{\texttt{M-}} = \begin{pmatrix}1&-1\\0&1\end{pmatrix} \quad \text{subtract display from memory}$$

$$\boxed{\texttt{AC}} = \begin{pmatrix}0&0\\0&1\end{pmatrix} \quad \text{clear display}$$

$$\boxed{\texttt{MRC}} = \begin{pmatrix}0&0\\1&1\end{pmatrix} \quad \text{recall memory}$$

$$\boxed{\texttt{MRC}}\boxed{\texttt{MRC}} = \begin{pmatrix}0&0\\1&0\end{pmatrix} \quad \text{recall and clear memory}$$

Other buttons on the calculator are equals, digits, and the usual operators for addition and subtraction, *etc.* I will not discuss them further in this paper.[7] However, for completeness we need a 'do nothing' or identity operation:

$$I = \begin{pmatrix}1&0\\0&1\end{pmatrix} \quad \text{do nothing}$$

Anything multiplied by I is unchanged. (This standard bit of notation is useful in much the same way that 0 is useful in ordinary arithmetic to represent nothing.)

Note that $\boxed{\texttt{MRC}}\boxed{\texttt{MRC}}$ is defined specially by Casio; it does not mean the same as $\boxed{\texttt{MRC}}$ pressed twice, which we can work out:

[7] Over the chosen two dimensional space, some operators are not linear (division and square root being obvious examples). Some keys require bigger matrices than we are using; see Appendix 8.1.

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

Thus, if MRC has the meaning as we defined in the matrix, then pressing it twice should have no further effect: the multiplication shows that apparently MRC MRC = MRC. But on the calculator, MRC MRC is instead defined to clear the memory after putting it in the display. Hence we defined a special matrix for it, but later we shall see how this ambiguity creates problems for the user — which the calculations above warn it may.

If MRC MRC does not correspond to the matrix for MRC, aren't I contradicting myself about the user doing matrix algebra? What it means is that our initial understanding of the user's model of the calculator, namely the state space $(d\ m)$, was inadequate. The state space should account for whether MRC is or is not the last button pressed. (It is possible to extend the state space to do this, but it becomes larger; see Appendix 8.1.) Casio seem, in effect, to think MRC MRC is a single operation that the user should think of as practically another button: in this sense we are justified in giving it an independent matrix (it also allows the rest of this paper to use 2×2 matrices rather than larger ones). However, the calculator does nothing to encourage the user to think of MRC MRC as a single button: pressing MRC then waiting an arbitrary time then pressing MRC again (as might happen if the user goes for a cup of tea) is treated as the special MRC MRC. In this case, the user would have no sense of MRC MRC being a single 'virtual' button.

In short, the problems I am glossing indicate a potential design problem with the HL-820LC: if we were designing a new calculator and had the ability to influence the design, we would have made different decisions.

What does MRC MRC MRC mean? Since Casio define MRC MRC to mean something special, then MRC MRC MRC could mean either (MRC MRC) MRC or it could mean MRC (MRC MRC) — which is the same thing the other way around. Provided we consider MRC MRC as a 'logical' button, this is an issue of commutativity. Instead, we could consider the matrix M for MRC directly, where our calculation shows $M(MM) \neq (MM)M$, and this would be a failure of associativity. But matrix multiplication *is* associative! Again, the problem this indicates is that M is bigger than a 2×2 matrix, and our calculations projected onto a 2×2 matrix are incorrect if we want to capture these idiosyncracies. Had we been designing a new calculator, rather than reverse engineering an existing one, the formal difficulty of representing MRC MRC as defined for this calculator might have encouraged finding a simpler design. Certainly it highlights a design issue that needs further exploration, whether empirical or analytic.

It would not matter if both of these alternatives had the same meaning. But as

$$\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} = \overset{\boxed{\texttt{MRC}}\boxed{\texttt{MRC}}}{\begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}} \overset{\boxed{\texttt{MRC}}}{\begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}} \neq \overset{\boxed{\texttt{MRC}}}{\begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}} \overset{\boxed{\texttt{MRC}}\boxed{\texttt{MRC}}}{\begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$$

we have established that the three successive key presses MRC MRC MRC is ambiguous: it could mean either (MRC MRC) MRC or MRC (MRC MRC) — and it matters

which! Thus the algebraic property of associativity is closely related to modelessness.[8]

When we look at the calculator to find out how Casio have dealt with this ambiguity, we find that we did not fully understand what MRC does. In fact, as one can establish with some experimenting: MRC only recalls the memory to the display if the memory is not zero; if the memory is zero, a single MRC does nothing. A double MRC MRC sets the memory to zero. What MRC MRC MRC means, then, is "recall memory to display then zero the memory": on the Casio this means exactly the same as MRC MRC does.

$$\boxed{\texttt{MRC}} = \begin{cases} m \neq 0, & \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} \\ m = 0, & \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \end{cases}$$

We now have a standard problem. The real calculator behaviour suggests we should extend our user model to handle what it actually does: the MRC button is not a simple 2×2 matrix! Or if we were Casio, we could redesign the calculator so that it has a simpler algebra — this is a route I'd prefer, but of course we cannot now change the HL-820LC. What I will do here in this paper is be careful that we never rely on doing MRC when the memory is zero. Since that is the simplest course for us, it is probably also the simplest course for a user. *The exception in the button almost certainly makes the calculator harder to use.* If the user is doing some calculations and repeatedly using M+ and M− they also and additionally have to keep track of whether the memory ever becomes equal to zero: if it does, pressing MRC will behave unexpectedly. If the memory is m, the user expects MRC to leave the calculator in the state $(m\ m)$, including $(0\ 0)$ if $m = 0$, but as Casio designed the calculator it will leave it in state $(d\ 0)$ in this case! Put another way, modes are messy (as we knew), and the matrix approach makes this very obvious.

In general, we have a useful design insight: if we can't capture a device's semantics easily, then possibly the device's design is at fault. Difficult semantics must mean, in some sense, that a device is harder to use than one with simpler semantics.

We can find some nice properties; we will look at just two.

First, MRC followed by M− sets the display to the memory and clears the memory. It is the same as Casio's interpretation of MRC MRC:

$$\underset{\begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}}{\boxed{\texttt{MRC}}} \quad \underset{\begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}}{\boxed{\texttt{M-}}} = \underset{\begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}}{\boxed{\texttt{MRC}}\boxed{\texttt{MRC}}}$$

This seems so simple, both equivalent solutions are the same length, so was the idiosyncratic interpretation of MRC MRC necessary to provide?

[8] Matrix multiplication is always associative, of course. Modes appear when the state space is too small, and associativity is maintained by having two or more matrices for what is a single user action. The trade-off is rather like a user would make: if they have an accurate model (i.e., their mental model state space is big) modes do not worry them; if they have too simple a model (i.e., their mental state space is missing some components), modes cause errors or erroneous reasoning.

Secondly, M+ and M− are inverses of each other:

$$\boxed{\texttt{M+}}\boxed{\texttt{M−}} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = I$$

Matrix algebra tells us immediately that pressing the buttons in the other order will have the same effect. We can also show this by explicit calculation:

$$\boxed{\texttt{M−}}\boxed{\texttt{M+}} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = I$$

So M+ is the inverse of M− and M− is the inverse of M+. If you press M+ by mistake, you can recover by pressing M− and *vice versa*.

By calculating determinants, it is easy to see that for the Casio calculator, none of the buttons apart from M+ and M− can be inverted. As we showed in Section 4, above, this means that a user cannot undo mistakes with any of the other buttons we have defined (of course a user would end up using digit keys and other features we have not mentioned here).[9]

6.1 Solving user problems: Task/action mappings

We have defined simple matrices for the memory buttons and the clear button. The Casio calculator can obviously add to memory (using M+) and subtract from memory (using M−). The question, now, is what else would we reasonably want a calculator like this with memory functions to do?

If the calculator's state is the vector $(d\ m)$ then plausibly we want operations to get to any of these states:

Zero display	$(0\ m)$
Zero everything	$(0\ 0)$
Zero memory	$(d\ 0)$
Show memory	$(m\ m)$
Store display	$(d\ d)$
Swap memory & display	$(m\ d)$

Most of these operations are easy to justify in terms of plausible user requirements. For example, solving a sum like $(4 + 5) \times (6 + 7) \times (8 + 9)$ requires the ability to save intermediate results in memory on this calculator that, like most basic calculators, does not have brackets. The final one, 'swap,' which seems more unusual, might be useful for a user who was not sure what was in the memory. One swap will confirm what is in the memory, and used again will put it back and restore the display as it was.

Conventionally, the problem is called *task/action mapping* [35]. The user's task is (for instance) to zero the display. Somehow the user has to map that idea of

[9] A careful reader will notice that the buttons I have defined do not allow the user to change the calculator's state from (0 0), so there is *technically* no problem if buttons have no inverse, because the calculator can't be got into other states anyway! In other words, our 2×2 model is too small to make all the points we'd like to from it. Even so, as designers we were able to spot the problem (which is an advantage), and it can be fixed.

a task into a sequence of actions; what they do is a task/action mapping. We have the advantage over a typical user in being able to specify the task and the actions as matrices. The task/action mapping can therefore be treated as a purely mathematical problem, that of solving some equations. Only if the mathematical analysis is trivial is the user interface going to be reasonably easy to use; if the mathematics is tricky, we probably have good reason to expect the user interface (or at least, some specific task/action mappings) will be very difficult — unless the user interface makes some concessions to the user that the mathematics hasn't captured (for instance, maybe the user manual or help system can provide a direct answer, thus avoiding the user having to do any further working out).

We can do some of the operations listed above very easily: for instance, AC achieves zero display (without changing the memory), as we noted earlier. Showing the memory is also done directly, but by MRC:

$$(d\ m)\,\boxed{\texttt{MRC}} = (m\ m)$$

because

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

and

$$(d\ m)\begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} = (m\ m)$$

To zero everything is fairly easy, since (with the special meaning of two consecutive MRC presses):

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

which will take $(d\ m)$ to $(0\ 0)$. However, doing these operations in the other order is *not* the same at all:

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

The operations do not commute; the user has to remember the right way round of using them. Indeed, doing MRC AC MRC is different again (as can be confirmed by calculating the product)! One might assume that if the user has to remember that MRC AC MRC and AC MRC MRC are different that the user interface is harder than necessary; it certainly ignores permissiveness [30].

The remaining three operations, storing the display, zeroing the memory and swapping, are a *lot* more tricky than these first few examples.

Imagine the user has the calculator displaying some number d and they want to get it into the memory. They must effectively solve this equation to find the matrix M, or a product of matrices equal to M if more than one button needs pressing:

$$(d\ m)\,M = (d\ d\,)$$

How ever a user might — or might not — find a solution (by guesswork? — though 'trial and error' is not feasible, since the calculator has no undo function), it is not difficult to solve this equation using matrix algebra:

$$(d\ m)\begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix} = (dM_{11} + mM_{21} \quad dM_{12} + mM_{22}) = (d\ d)$$

so $M_{11} = 1, M_{21} = 0$ and so on. Putting it all together we get

$$M = \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}$$

This is not one of the matrices we have got directly available. No button press corresponds to M. Of course users (apart from us!) don't solve matrix equations explicitly, instead they will have to do some rough work and hard thinking. In the examples here, the complexity suggests it is very likely that no users (except mathematicians) would be able to work out how to do things — the Casio appears to be far too complex *if we think the tasks listed above are reasonable for it to support.*

Fortunately, we have the huge benefit of having a handy design notation which makes things much easier to work out. Once we have worked out a solution, we might say how to use it to solve the problem in the user's manual: a user does not need to go over all the work again. Alternatively the calculator could be redesigned so that it had a button that did M directly: there would then be little need to explain it in detail in the user manual (or on the back of the calculator). In this case, of course, we'd need to be satisfied that the feature was sufficiently useful that it was worth dedicating a button to. Another possibility (which we'll see again below) is that it might be possible to choose other functions on the calculator to make working out M a lot easier.

Given that the Casio calculator design is fixed, we shall have to work M out as a product from some or all of the matrices we already have. (If Casio had provided a key with a corresponding matrix M things would have been trivial.) It cannot be done with the keys we have defined so far. We need to use, for example, the [−] and [=] keys too. We can work out (which as we have seen with [MRC] and other keys, needs carefully checking by experiment as well) that

$$[-][\mathtt{MRC}][=] = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}$$

with this insight,[10] we can establish with routine work (which was done by *Mathematica* [34]) that

$$[-][\mathtt{MRC}][=][\mathtt{M+}][\mathtt{MRC}] = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}$$

[10] Fortuituously this particular sequence of keystrokes only requires a 2×2 matrix! *This simple matrix definition will fail if there is numerical overflow* — because the calculator gets 'stuck' when there is an error, and our current two-component state space cannot model this feature.

as required.[11] If it's so difficult — both to work out and to do — why would we want to do it? Simple: the calculator has a memory and we might want to store a number we have worked out, currently in the display, into the memory. Surely that is what the memory is for?

It seems clear that the calculator should have provided a STORE key to do this operation directly. It would then be very easy to store the display in memory. Note that with the Casio design as it is, this sequence of operations includes matrices that are not invertible: if the user makes a mistake, there is no way to recover (unless the user knows what the state previous was and how to reconstruct it). One needs to use a piece of paper to keep a record of the calculations — and if you're using a piece of paper, what real use is the memory?

Next, for the user to get the task represented by $(m\ d)$ done, which is just swapping over the display and memory, we need to find factors of M in terms of our existing matrices, where

$$M = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

We might want a swap operation so we could work on two calculations at once, one in the display and one 'backed up' in the memory. (We could be keeping track of two tallies.) Another use of a swap is to allow the user to perform any calculation on the memory without losing the displayed number: for example, with a swap the user could square root the memory using the standard square root button, and no special memory-root button would be required.

Now this M is invertible — a swap followed by a swap would leave the calculator's display and memory unchanged — so it cannot be a product of any of the existing matrices except M+ or M−, which are the only invertible matrices. Since M+ and M− commute, for any sequence of using them (with no other buttons used between them) their order does not matter. So, to find out what an arbitrary sequence of using them means, we can collect them together, say putting all the M− first. The most general sequence is then

$$\overbrace{\boxed{\texttt{M-}}\boxed{\texttt{M-}}\cdots\boxed{\texttt{M-}}}^{m\text{ times}}\overbrace{\boxed{\texttt{M+}}\boxed{\texttt{M+}}\cdots\boxed{\texttt{M+}}}^{n\text{ times}} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}^m \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}^n = \begin{pmatrix} 1 & n-m \\ 0 & 1 \end{pmatrix}$$

Note that $\boxed{\texttt{M+}}^n = \boxed{\texttt{M-}}^{-n}$ a fact that we will use later. Hence $\boxed{\texttt{M+}}^n$ can never be M, for any n, positive or negative; we have proved that the swap operation is impossible on the Casio. Incidentally, as a by-product of this line of thought, we now have a formula that enables us to find out how to do any task on the calculator that requires a matrix

$$\begin{pmatrix} 1 & n \\ 0 & 1 \end{pmatrix} \text{ for any integer } n$$

[11] There are many other solutions, but this is one of the easier ones. There are shorter solutions for the real calculator, but their derivation involves knowing undocumented details of its operation that are beyond the scope of this paper.

Obviously Casio could provide a special SWAP key which does what we want in one press, but it is creative to ask what else could be done. First we prove a swap would have to be achieved in combination with using M− or M+, assuming no other buttons than those we have so far defined are available.

If the new button has matrix S and it helps the user achieve a swap, then it must be the case that there are matrices A and B such that

$$ASB = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

This is not singular (its determinant is -1), and therefore the determinants of A, S, and B are all non-zero. To solve the equation for S, we get:

$$S = A^{-1} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} B^{-1}$$

A special case is that A and B are both identities, and then S is, of course, the swap matrix itself. If A and B are not singular, then they cannot be products of singular matrices: in short, on the Casio, they can only be composed out of M− and M+.

For example if we wanted a new button S that did a swap when used between M+ and M−, we would solve this equation

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} S \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

so

$$S = \begin{pmatrix} -1 & 0 \\ 1 & 1 \end{pmatrix}$$

In words, the button S (as defined here) has an English meaning, "Subtract display from memory and display it."

Perhaps we could try $A = I$ and $B =$ M−, to find the operation that when followed by M− gives a meaning equivalent to SWAP. We then need to solve

$$S \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

which gives S the English meaning, "Add the display to the memory, and display the original memory."

A more general approach is possible. Earlier I gave the general form for any number of M− or M+ used together, namely $\boxed{\texttt{M+}}^n$ for positive or negative n. We can solve the equation for $\boxed{\texttt{M+}}^m S \boxed{\texttt{M+}}^n = \boxed{\texttt{SWAP}}$ and get

$$S = \begin{pmatrix} -m & 1+mn \\ 1 & n \end{pmatrix}$$

In other words, there are no forms that would give a brief and concise natural interpretation for a new button S. Moreover, any meaning we might have liked for

S can be achieved using some combination of SWAP, M− and M+: if we have SWAP we do not need any of these esoteric buttons.

Overall, then, it would be better to have a SWAP button, or collect persuasive empirical evidence that users do not want a swap operation! Since there are many calculators on the market with the same style of design (i.e., memory buttons but no memory store button) and have been for several years, it would seem that users find the design sufficiently attractive. This might mean that the memory buttons are there to sell calculators, or it might mean that users do not do sophisticated things with calculators, ... leading us into speculation beyond the scope of this paper.

Finally, consider the zero memory task. Here we need to find a matrix M such that

$$(d\ m)\,M = (d\ 0)$$

The Casio has no key that does this directly, so — as before — we will have to find a sequence of button presses B1 B2 $\cdots$ Bn whose matrices multiply together to make M. If we had a STORE key, all this could have been achieved by doing STORE MRC MRC: the STORE stores the display in memory, then the double-MRC recovers memory and sets it to zero. We can check our reasoning:

$$\underset{\begin{pmatrix}1&1\\0&0\end{pmatrix}}{\boxed{\texttt{STORE}}}\ \underset{\begin{pmatrix}0&0\\1&0\end{pmatrix}}{\boxed{\texttt{MRC}}\boxed{\texttt{MRC}}} = \begin{pmatrix}1&0\\0&0\end{pmatrix}$$

and

$$(d\ m)\begin{pmatrix}1&0\\0&0\end{pmatrix} = (d\ 0)$$

Although the Casio does not have a STORE button, we have already worked out a sequence that is equivalent to it, namely − MRC = M+ MRC. So to put zero in the memory, we should follow it by MRC MRC, which would mean this:

− MRC = M+ MRC MRC MRC

but this has got that problematic sequence of three consecutive MRC presses we examined earlier. We discovered that the Casio would treat this as meaning the same as

− MRC = M+ MRC MRC

Coincidentally this gives the required matrix, exactly what we wanted. There is no shorter solution.[12]

Earlier I claimed that the potential ambiguity of MRC MRC MRC could cause problems. Yet here it looks like Casio's design helps. Actually, we did not *know* Casio's design decision would help — we had to work it out by doing the relevant matrix multiplications. In other words, to find out how to perform a trivial and plausible-sounding task, we had to engage in formal reasoning that is beyond most users. This strongly suggests the calculator is badly designed in this respect.

[12] With the usual provisos. The device does have some undocumented tricks I have not exploited.

Add display to memory	M+
Subtract display from memory	M−
Show memory	MRC
Zero display	AC
Zero memory	− MRC = M+ MRC MRC
Zero everything	MRC MRC AC
Store display	− MRC = M+ MRC
Swap memory & display	*impossible*

Table 1. How the Casio HL-820LC can be used to do memory operations (with shortest solutions shown).

6.2 Summary and comparisons with other designs

Table 1 summarises our results for the Casio HL-820LC. There is no inevitablility about these results: Casio themselves make other calculators that embody different design decisions.

The Casio HS-8V has a change sign button, whic the HL-820LC does not:

$$\boxed{+/-} = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \text{ change sign}$$

Since this button is invertible, it provides more ways of attempting to factor the swap operation. Indeed,

$$\boxed{M+}\boxed{-}\boxed{MRC}\boxed{M+}\boxed{+/-} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$\boxed{M+}\boxed{+/-}\boxed{MRC}\boxed{-}\boxed{=} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

are swaps. This is clearly about as hard to do as storing the display in the memory (on either the HL-820LC or this, the HS-8V), and the same design and usability comments apply.

The Casio MS-70L does not have an MRC button, instead having an MR button. The matrix for this is the same as a single press of the HL-820LC's MRC:

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

Further, MR MR = MR: no special meaning is attached to repeated use of this button (which is what we expect from its matrix definition above). As noted earlier in this section, the HL-820LC's idiosyncratic meaning for MRC MRC can be achieved on the MS-70L by MR M−. In other words, there does not seem to be an intrinsic reason why the HL-820LC was designed so confusingly.

The Casio MS-8V has the MS-70L buttons together with a MC as a single key, and it has a button MU which has nothing to do with memory but is a kind of percent key, mark up.

The different Casio designs, MS-70L and MS-8V, both have much simpler matrix semantics than the HL-820LC or HS-8V, and one might therefore hypothesise they

are much easier to use reliably. However, they both share with the HL-820LC the absence of the STORE button.

7. EXAMPLE 3: THE FLUKE 185 DIGITAL MULTIMETER

In the previous examples, we considered an exact model of part of a device (the Nokia mobile phone, Example 1), and we considered an abstract model of device (the Casio calculator, Example 2). The abstract model of the calculator ignored some user actions (such as the multiply function), to concentrate on the use and behaviour of display and memory in handling numbers. We now do the opposite: we consider *every* function of a device, but abstract away from the numbers it represents for the user. If the example was a CD player, we might be abstracting away the track number and the time playing a track, but this example is a digital multimeter, and the values abstracted away are voltages, resistances and so on. Unless we are interested in the legibility of the numeric display (or teaching users to read numerals) the actual numbers are not important for device-level usability issues[13] — whereas for the calculator correctly handling specific numbers was crucial to achieving task goals correctly.

Our final example, then, is a digital multimeter, an electrical measuring instrument. The Fluke 185 has ten buttons, four of which are soft buttons with context-sensitive meanings, and a rotary knob with seven positions. It has a 6.5×5cm LCD screen with about 50 icons, as well as two numeric displays and a bar graph. It has the most complex user interface of the examples considered in this paper. Furthermore, it is a safety critical device: user errors with it can directly result in death. For example, if the Fluke 185 meter is connected to the AC mains electricity supply but set to read DC volts, it will not warn that the voltage is potentially fatal. As a general purpose meter, it can be connected to sensors, such as the Fluke carbon monoxide sensor, and a misreading could lead to gas poisoning — if the meter is set to AC volts it would read 0 whatever the DC voltage from the sensor. When measuring amps the meter, cables and the device being tested could overheat and cause a fire (or even an explosion). And so on; user/design error can lead to immediate and possibly catastrophic physical consequences.

A multimeter should only be used by competent users, and therefore the trade-offs between usability and interface complexity are different than general use products (such as mobile phones). However, unlike a professional interface (such as an aircraft cockpit), multimeters may be used by users competent in the domain but who do not fully understand or have forgotten details of the user interface. Thus the user interface should not have many modes, feature interactions, timeouts or other features that are not essential in the domain; where possible it should utilise warnings, utilise interlocks, and be self-explanatory, *etc.* Such issues are very important and require careful consideration, based on a deep understanding of the domain (both technical and regulatory) and of human error and so on. These issues go beyond matrix algebra and this paper is silent on them, beyond noting the general point that there is a danger that technical concerns take precedence over user interface

[13] If the user's tasks were ecological, say correctly measuring resistances in a circuit, we could have matrix elements that represented 'correct' and 'incorrect' (say) rather than the precise values. However such considerations would take us too far from the core of the paper.

concerns, and that user interface quality will suffer. As a case in point, the Fluke 185 conforms to seven types of safety standard and a quality standard (ISO9001), but to no user interface standard.

7.1 Specifying the Fluke 185

Since the Fluke 185 is very complex, a thorough specification would be very tedious, but I will show how all its various feature types can be handled using matrices.

The Fluke's rotary knob has the nice feature that its meaning depends only on its current position. A rotation to 'off' can be modelled by OFF, as if it was an independent button.

Some devices use knobs differently. If the knob was on a safe, then the security of the safe would depend on sequences of knob positions (used to enter combination codes), and we would need to model knob positions as pairs, such as 7→3 or (on the meter) V→OFF if it is turned to off from the volts position. However for the Fluke under consideration, this complexity is not necessary — and as a result of having fewer matrices (n rather than n^2 for a knob of n positions) and a smaller state space (since there is no need to keep track of previous knob positions), the meter is far easier to use than trying to break into a safe!

The next feature to model is that the meter can be off or on. When it is off, no button does anything, and the knob has to be turned away from the off position to turn it on. Assume the off state is state 1; then the state vector $(1 \vdots \mathbf{0})$ represents the meter off, and the state vector $(0 \vdots \mathbf{s})$ represents it in any state in the subvector $\mathbf{s}$ other than off. (This is a partitioned state space; see Appendix A.1.)

For generality, represent any button (or knob position) other than OFF by B. Let B represent the effect of any such button when the meter is on (so $\mathbf{s}B$ is the effect of button B on state $\mathbf{s}$ when it is on), then the full matrix for each B that correctly extends to an on/off meter is the partitioned matrix

$$\left(\begin{array}{c:c} 1 & \mathbf{0} \\ \hdashline \mathbf{0} & B \end{array}\right)$$

If the meter is in state $(x \vdots \mathbf{s})$, this partitioned matrix transforms it to state $(x \vdots \mathbf{s}B)$. Thus if it was off $(x = 1, \mathbf{s} = \mathbf{0})$ it is still off; if it was on $(x = 0)$ it is still on and the on state $\mathbf{s}$ has been transformed as expected to $\mathbf{s}B$. Note that no button B embedded in a matrix of this form can switch the meter off.

The off switch takes every state to off, and its matrix would have the particularly simple structure as follows (here written explicitly for a FSM with 6 states, including off, which is state 1):

$$\text{OFF} = \left(\begin{array}{c:ccccc} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

This takes any state $(x \mathbin{:} \mathbf{s})$ to $(1 \mathbin{:} \mathbf{0})$ as required. Repeating off does not achieve anything further; we can check by calculating $\boxed{\texttt{OFF}}\,\boxed{\texttt{OFF}} = \boxed{\texttt{OFF}}$, which is indeed the case. Mechanically, because $\boxed{\texttt{OFF}}$ is a knob position, it is impossible to repeat off without some other intermediate operation: thus it is fortunate it is idempotent and the user would not be tempted to 'really off' the meter! (Compare this situation with the $\boxed{\texttt{AC}}$ key on the calculator.) In short, the knob's physical affordance and its meaning correspond nicely [31].

In summary, once we have explored the meaning of $\boxed{\texttt{OFF}}$ and the general behaviour of other buttons whether the meter is on or off, as above, we can ignore the off state and concentrate our attention on the behaviour of the meter's buttons when it is on. Partitioning is a powerful abstraction tool, and henceforth we can assume the meter is on.

The meter has a display light, which can be switched on or off. The behaviour of the meter is otherwise the same in either case. Introducing a two-state mode (light on, light off) therefore doubles the number of states. If the 'light off' states are $1 \ldots N$, then the 'light on' states are $(N+1) \ldots 2N$, a simple way of organising the state space is that switching the light on changes state from s to $s+N$, and switching it off changes the state from s to $s-N$.

Now represent the state vector by $(\mathbf{off} \mathbin{:} \mathbf{on})$. The light switch can be represented by a partitioned matrix L that swaps the on and off states:

$$L = \left(\begin{array}{c:c} \mathbf{0} & I \\ \hdashline I & \mathbf{0} \end{array}\right)$$

The submatrices I and $\mathbf{0}$ here are of course all $N \times N$ matrices. This works as required, as can be seen by multiplying out the effect of L on a general state vector

$$(\mathbf{x} \mathbin{:} \mathbf{y}) \left(\begin{array}{c:c} \mathbf{0} & I \\ \hdashline I & \mathbf{0} \end{array}\right) = (\mathbf{y} \mathbin{:} \mathbf{x})$$

For any other operation H (such as pressing, say, the $\boxed{\texttt{HOLD}}$ button) we want its effect in the light off states and the light on states to be the same. Again this is achieved using partitioned matrices, but in the following form:

$$\left(\begin{array}{c:c} H & \mathbf{0} \\ \hdashline \mathbf{0} & H \end{array}\right)$$

Here, the submatrix H in the top left is applied to the light off states, and the submatrix H in the lower right is applied to the light on states. For example, whatever state the meter is in with the light off, it goes to $\mathbf{off}H$ as required; and if the meter's light is on, then $\mathbf{off} = \mathbf{0}$ and $\mathbf{off}H = \mathbf{0}$. From this matrix, it is clear that H behaves as intended, identically in either set of states, and in particular it does not switch the light on or off. Thus matrices easily model the behaviour of all buttons.

This recipe for handling any matrix H is independent of the light switch matrix. Thus we can consider the design of the meter ignoring the light, or we can consider

the light alone, or we can do both. As before, this freedom to abstract reliably is very powerful. Indeed, if we were trying to reverse engineer the complete FSM for the meter, building it up from the matrices in this way for each button would be a practical approach.

Given the partitioned structure of the meter's button matrices, it is easy to show that the light switch matrix L commutes with all matrices (except OFF, which requires L to be a submatrix of itself; see also §7.2), which means a user can start a measurement and then switch the light on, or first switch the light on — the results will be the same. Other modes, such as measurement hold, can be treated similarly.

At switch on, the meter allows 12 features to be tested, activated or adjusted. For example, switching on while pressing the soft F4 key causes the meter to display its internal battery voltage. (Because this is a switch-on feature, the soft key cannot be labelled when the defice is off; a user would need to remember this feature.) As so far described, this is easy to model: the meter has one extra state for this feature, and it is reached only by the sequence OFF V F4, where V F4 represents a simultaneous user action, turning the knob to volts while pressing F4. Its matrix is routinely defined from submatrices. However, as the Fluke meter is defined, the light button does not work in the new state: rather than switching on the light, it exits the new state and puts the meter into normal measuring mode (i.e., the state corresponding to the normal meaning of whatever the knob is set to). If we are to model this feature, the elegant 'law' about the light matrix and other button matrices needs complicating: another idiosyncratic user interface feature is hard to model in matrix algebra. Although I do not have space to go into details here, this awkwardness with matrix representation seems to be an advantage:[14] had Fluke used a matrix algebra approach, they would have been less likely to have implemented this special feature. Indeed, the meter has a menu system (with 8 functions) that can be accessed at any time: why wasn't the battery voltage feature put in the menu, so its user interface was handled uniformly?

The Fluke meter has a shift button, which changes the meaning of other buttons if they are pressed immediately next. (It only changes the meaning of three buttons, including itself, all of which anyway have extra meanings if held down continuously; additionally, the shift button has a different, non-shift, meaning at switch on.) In general if S represents a shift button and A any button, we want SA to be the button matrix we choose to represent whatever "shifted A" means, and this should depend only on A.

For any button A that is unaffected by the shift, of course we choose $SA = A$. Since the shift button doubles the number of states, we can define it in the usual way as a partitioned matrix acting on a state vector (**unshifted-state** ⋮ **shifted-state**). Since (at least on the Fluke) the shifted mode does not persist (it is not a lockable shift), all buttons now have partitioned matrices in the following simple form

[14] It requires a 3×3 partitioning, and no longer has the nice abstraction property; see §7.4 for other examples.

$$\left(\begin{array}{c:c} A_{\text{unshifted}} & \mathbf{0} \\ \hdashline \mathbf{0} & A_{\text{shifted}} \end{array}\right)$$

and

$$S = \left(\begin{array}{c:c} \mathbf{0} & I \\ \hdashline I & \mathbf{0} \end{array}\right)$$

which (correctly, for the Fluke) implies pressing SHIFT twice leaves the meter unshifted (since the submatrices are all the same size and $SS = I$).

For a full description of the meter, the various partition schemes have to be combined. For example, we start with a matrix R_{basic} to represent the RANGE button (the RANGE button basically operates over 6 or so range-related states, the exact number depending on the measurement), and we build up to a complete matrix R which represents the RANGE button in all contexts. As it happens R shifted is equal to R, and we can define the shift matrix:

$$R_s = \left(\begin{array}{c:c} R_{\text{basic}} & \mathbf{0} \\ \hdashline \mathbf{0} & R_{\text{basic}} \end{array}\right)$$

Next we extend R_s to work with the light button. Again, R is the same whether the light is on or off, so we require:

$$R_{sl} = \left(\begin{array}{c:c} R_s & \mathbf{0} \\ \hdashline \mathbf{0} & R_s \end{array}\right)$$

Now R_{sl} defines the subset of the RANGE button's meaning in the context of the additional features activated by the shift key and the light keys. Note that $R_{sl} = R_{ls}$ (the meaning is independent of the order in which we construct it) as expected — but see qualifications in §7.2. Next we extend R_{sl} so R works in the context of the meter being on or off:

$$R = \left(\begin{array}{c:c} 1 & \mathbf{0} \\ \hdashline \mathbf{0} & R_{sl} \end{array}\right)$$

This completes the construction of R with respect to the features of the meter I have defined so far. Incidentally, this final matrix illustrates a case where the submatrices are not all square and not all the same size. If R_{basic} is a 6×6 matrix, this R is 25×25.

7.2 Qualifications

Sadly I idealised the Fluke 185, and the real meter has some complexities that were glossed in the interests of brevity. The main qualifications are described in this section.

I claimed that the construction of the R matrix to represent RANGE was independent of the order of applying shift and light constructions. Although this is true as things are defined in this paper, it is not true for the Fluke 185, a fact which I discovered as soon as I checked the claim! The light and shift buttons do not commute on the Fluke: SHIFT LIGHT $\neq$ LIGHT SHIFT. This quirk can only make the Fluke harder to use (it also makes the user manual a little less accurate, and it will have made the meter harder to program).[15]

In fact, the RANGE button has different effects under different measurement conditions, a different effect at power on, and it has a different effect in the menu and memory subsystems, and another effect when it is held down for a few seconds. The range feature also interacts with min/max; for example, it can get into states *only* in min/max mode where autoranging is disabled. For simplicity, I ignored these details, which would be handled in the same way (but unfortunately not as cleanly) as the example features. Such feature interactions make the matrix partitions not impossible, but too large to present in this paper.

Indeed, the Fluke 185 has many feature interactions. It seems plausible to conclude that Fluke put together a number of useful features in the meter considered independently but did not explore the algebra of the integrated user interface. The user interface, when formalised, therefore reveals many *ad hoc* interactions between features, all of which tend to make the user interface more complex and harder to use, and few or none of which have any technical justification.

The Fluke also has time dependencies. These can all be handled by introducing a new matrix τ that is considered pressed every second, however the result is mathematically messy and not very illuminating. Had the designers specified the user interface fully and correctly, they would naturally have wanted to avoid such messiness; furthermore, I can see no usability benefits of the timeouts in this context, to avoid the mathematical messiness would have improved the user interface design.

The Fluke has a few further user interface features, such as soft buttons (the meaning of a soft button depends on what the display panel shows). These features provide no difficulties for matrix algebra, and three following sections (§7.3, 7.4 and 7.5) complete the example.

7.3 Reordering states

To handle the remaining features, we need to review a standard matrix technique; the benefit is further insight into tradeoffs in user interface design.

Section 7.1 made repeated use of partitioned matrices and made claims about the relevance of the structure of matrices to the user interface. However it is not immediately obvious that as partition structures are combined (e.g., for on/off and other features) that any relevant structures can be preserved, or even seen to be preserved in the form taken by the matrix structures. This section briefly considers this issue, and shows that it is trivial.

[15] If the meter's software program is modularised, this 'quirk' requires wider interfaces or shared global state, both of which are bad programming practice and tricky to get correct. If the meter's program is not modular, then any *ad hoc* feature is more-or-less equally easy to program; but this is not good practice: because there is then no specification of the program other than the *ad hoc* code itself.

A user need not be concerned with state numbering, and in general will have no idea what states are or are numbered as. From this perspective we can at any time renumber states to obtain any matrix restructuring we please, provided only that we are methodical and keep track of the renumberings. Whilst this statement is correct, it is rather too informal for confidence. Also, an informal approach to swapping states misses out on an important advantage of using matrices to do the job cleanly.

Any state renumbering can be represented as consistent row and column exchanges in a transition matrix. For example, if rows 2 and 3 are swapped, and columns 2 and 3 swapped, then we obtain the equivalent matrix for a FSM but with states 2 and 3 swapped. In general any renumbering is a permutation which can be represented by a matrix P. If M is a matrix, then PMP^{T} is the corresponding matrix with rows and columns swapped according to the permutation P. If several permutations are required, they can either be done separately or combined, whichever is more convenient: e.g., as $P_1P_2MP_2^{\mathsf{T}}P_1^{\mathsf{T}}$ or as $P_3MP_3^{\mathsf{T}}$ where $P_3 = P_1P_2$.

In short, throughout this paper, when any matrix M was presented, implicitly a permutation P was chosen to present it cleanly: we wrote down a neat M to represent PMP^{T}.

7.4 Remaining major features

Consider the Fluke 185 autohold feature. Often a user cannot both look at the meter and handle the probes. The Fluke 185 provides two features to help: a HOLD button freezes the display when it is pressed, and the AUTOHOLD feature (holding down the HOLD button for a couple of seconds) puts the meter into a mode where it will automatically hold any stable non-zero measurement.

As so far described, modelling this requires a 2×2 partitioned matrix: the meter has two sets of state, normal ones and autohold ones. The AUTOHOLD simply swaps between the two, in the same way as the light on/off button:

$$A_{\text{basic}} = \left(\begin{array}{c:c} \mathbf{0} & I \\ \hdashline I & \mathbf{0} \end{array}\right)$$

Pressing the button again gets the meter out of autohold mode, and because both submatices are identities, returns the meter to the original state.

In fact, autohold is not available in capacitance measurements.[16] There is therefore one state where the autohold leaves the meter in the same state, namely the capacitance state. Introduce a permutation P so that the capacitance state is the first state, as this allows us to write down the autohold matrix in a particularly simple form:

[16]The meter may not implement autohold for capacitance possibly because the meter is unable to measure capacitance fast enough, and incomplete readings might be confused by the meter for stable readings. On the other hand, because capacitance measurements can be slow — they can take minutes (because high value capacitors charged to high voltages have to be discharged safely) — users would appreciate the autohold's automatic beeping when the measurement was ready.

$$A_{\text{capacitance}} = P \left(\begin{array}{c:c:c} 1 & \mathbf{0} & \mathbf{0} \\ \hdashline \mathbf{0} & \mathbf{0} & I \\ \hdashline \mathbf{0} & I & \mathbf{0} \end{array} \right) P^{\mathsf{T}}$$

This is essentially the same matrix as before, except state 1 goes to state 1 (courtesy of the top left 1). One identity has lost a row (and column) because it no longer takes capacitance to autohold-capacitance; the other identity has lost a row (and column) because the model no longer has an autohold-capacitance state to be mapped back to capacitance.

In fact, there are several states that are affected like this: capacitance has 9 ranges, covering 5nF to 50mF and an automatic range. Now let P permute the 9 capacitance states we are modelling to consecutive states 1 to 9. The single 1 of the matrix above that mapped a notional capacitance state to itself now needs to be generalised to an identity that maps each of the nine states to themselves. Of course this is a trivial generalisation as the relevant states are consecutive:

$$A_{\text{capacitance-ranges}} = P \left(\begin{array}{c:c:c} I & \mathbf{0} & \mathbf{0} \\ \hdashline \mathbf{0} & \mathbf{0} & I \\ \hdashline \mathbf{0} & I & \mathbf{0} \end{array} \right) P^{\mathsf{T}}$$

Here the top left identity submatrix is 9×9. Since the permutations P — being permutations of arbitrary state numberings — do not typically add a great deal to our knowledge of the system, we may as well omit them, and this is what we did elsewhere in this paper.

We can make a usability comment on this aspect of the user interface design of the Fluke 185. The matrix $A_{\text{capacitance-ranges}}$ has more information in it than the matrix A_{basic} (even allowing for A_{basic} to range over all nine states). Understanding the meter as designed apparently imposes a higher cognitive load on the user; also the user manual should be slightly larger to explain the exception (in fact, the Fluke manual does not explain this limitation). It is strange that autohold is not available for capacitance (perhaps increasing its time constant if necessary), since on those possibly rare occasions when the user would want it, it would presumably be both extremely useful to have it and extremely surprising to find it is not supported.

7.5 Soft buttons

The Fluke multimeter has soft buttons, and I have now reviewed appropriate matrix methods to handle them. The Fluke has four soft buttons, for example pressing F1 behaves variously as VERSION, AC, BLEEP, Ω or °C. In some set of states, F1 means AC and in another set of states it means BLEEP, and so on.

Let P be a permutation of states that separates these sets of states. The matrix for F1 can then be written clearly:

$$\boxed{\texttt{F1}} = P \begin{pmatrix} \boxed{\texttt{VERSION}} \\ \hdashline \boxed{\texttt{AC}} \\ \hdashline \boxed{\texttt{BLEEP}} \\ \hdashline \boxed{\Omega} \\ \hdashline \boxed{^\circ\texttt{C}} \end{pmatrix} P^\mathsf{T}$$

We may be interested in, say $\boxed{\Omega}$ alone, but it is a rectangular matrix. We can define square matrices like

$$\Omega = Q \begin{pmatrix} I & \vdots & \mathbf{0} \\ \hdashline \multicolumn{3}{c}{\boxed{\Omega}} \end{pmatrix} Q^\mathsf{T}$$

and this definition of Ω effectively means the same as a $\boxed{\Omega}$ button that is available at all times, but when pressed in states where (as a soft button) it would not have been visible it does nothing, by its identity submatrix.

8. ADVANCED ISSUES AND FURTHER WORK

Matrix algebra applied to user interface design opens up many new possibilities. This section introduces some more advanced topics, as well as new topics that require working out: raising issues that that cannot be solved without further research. In particular, we discuss some criticisms and possible oversights.

8.1 Button modes

In order to make for a clear exposition in Example 2, we relied on a small matrix to explore a calculator design, but occasionally I admitted the small model was inadequate for some details. This section makes clear that the self-imposed limitations are avoidable.

Modes require bigger matrices, but their structure is not very illuminating if they are based on a conventional flat FSM (as witnesss the lengthy discussions in Example 3). Instead, as this Appendix shows, matrices can be derived from hierarchical FSMs, where each 'state' of a higher level FSM represents a complete mode of the system, so a mode is an entire FSM itself, and represents how the system behaves within *that* mode.

We will illustrate how to build matrices by considering the $\boxed{\texttt{MRC}}$ button on the Casio calculator, which is modey. The $\boxed{\texttt{MRC}}$ button means different things depending on how often it is pressed. This is a typical sort of feature of many user interfaces. In the discussion of the calculator example (2).

The digital multimeter example (3) provided the tools to construct matrices for user actions like $\boxed{\texttt{MRC}}$. We now utilise the partitioning techniques to show that it is possible to build a matrix correctly modelling $\boxed{\texttt{MRC}}$ over a more intuitive state space based on (display memory), that is $\mathbb{R}^2$, rather than the Boolean spaces used for Example 3.

The calculator has three high-level modes:

(1) Button $\boxed{\texttt{MRC}}$ not used. This is the normal mode.

(2) Button MRC pressed once. The memory is recalled to the display. Call this button matrix M_1.

(3) Button MRC pressed more than once. The memory is cleared. Since this mode follows Mode 2, the display will continue to show the previous memory contents. Call this button matrix M_2.

The key MRC goes between the modes (specifically, $1 \to 2 \to 3 \to 3 \cdots$), and what we might call ANY (i.e., any action other than MRC) returns to normal mode ($\cdots \to 1$), as well as doing whatever it does in that mode. For example, AC returns to Mode 1 and sets the display to zero, whereas MPLUS adds the display to memory, but also returns to Mode 1.

If we proceeded from this specification, and rigorously followed the 'make a matrix recipe' we describe below, we would end up with a button matrix for MRC that was 9×9, but it would have three pairs of its rows equal! Rows being equal mean that some of the states are the same. Indeed, the definition of the modes above is misleading; what were called modes 2 and 3 are in fact the same. The description above put meaning into the different *modes* (or states) but the algebraic idea is to attach meaning to the *actions* — the button matrices.

A correct specification still requires the *two* matrices M_1 and M_2, but only two modes are needed: 'normal' (Mode 1) and 'MRC hit' (Mode 2). We thus now have a two state higher level FSM, as shown in Figure 2. The state vector will have the 2-component structure

$$\left(\begin{array}{c:c} \textit{Mode 1} & \textit{Mode 2} \\ \text{Normal} & \text{MRC hit} \end{array}\right)$$

The abstract mode behaviour of buttons for any operation other than MRC is captured by

$$\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}$$

since (with 1s in column 1) they keep or put the device in Mode 1. For MRC the abstract mode action is represented by

$$\begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}$$

since MRC takes the device to Mode 2 (i.e., the 1 in row 1), and keeps it there (i.e., the 1 in row 2).

This abstract structure is now refined by replacing each 'change mode' element 0 or 1 with either a zero matrix or the appropriate concrete matrix that operates *within* each mode. Thus if the button matrix for any action other than MRC is A then the partitioned matrix

$$\left(\begin{array}{c:c} A & \mathbf{0} \\ \hdashline A & \mathbf{0} \end{array}\right)$$

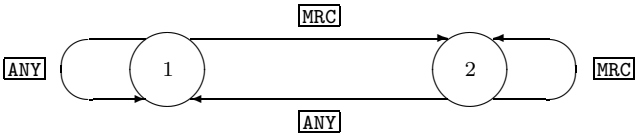


Fig. 2. The basic mode transitions.

represents the overall action A on the concrete two component space: it does the same thing in either mode. If A is 2×2, this partitioned matrix is a 4×4 matrix that effectively applies the transformation A in any mode, but always maps the result to some state in Mode 1.

Similarly, the button matrix for MRC has partitions

$$\left(\begin{array}{c|c} \mathbf{0} & M_1 \\ \hline \mathbf{0} & M_2 \end{array}\right)$$

Unlike the composite A matrix, this does different things in each of the modes. This matrix effectively applies the transformation M_1 in normal mode, Mode 1, and maps the result to Mode 2. In Mode 2, however, M_2 applies the second meaning of MRC and maps the result back to Mode 2. Thus M_1 is the button matrix for MRC pressed once (after anything other than MRC), and M_2 is for MRC pressed immediately after MRC (including any number of times).

In a simple Boolean-based state space, exactly one element of the state will be True, corresponding to the state the device is in, and the matrices and state space they operate on will be unambiguous. (This is how we treated partitioned matrices in Example 3.) But if the basic state space is taken to be $\mathbb{R}^2$, explicitly showing the numbers in the display and memory, there is no way in that space to indicate the current mode out of the choice of two modes. Thus we need to introduce a flag to specify which mode the calculator is in. Each mode, then, is represented by three components: 0 or 1, depending on whether this mode is active; the display value d; and the memory value m. The explicit state vector, in full, is then: $(\text{mode}_1\ d_1\ m_1 \quad \text{mode}_2\ d_2\ m_2)$, where mode_1 and mode_2 are 0 or 1 depending on whether the mode is active, and the following $d_i\ m_i$ pair is used when the corresponding mode_i is 1 (and otherwise ignored).

The sub-matrices now need to be extended to handle the mode flag, and this is a matter of putting a 1 in the appropriate place. For the two (mode-dependent) MRC matrices M,

$$M_i \mapsto \left(\begin{array}{c|c} 1 & \mathbf{0} \\ \hline \mathbf{0} & M_i \end{array}\right)$$

For the simple $(d\ m)$ space used throughout Example 2, with $M_1 = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$ and $M_2 = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$, we obtain a 6×6 matrix

$$\boxed{\texttt{MRC}} = \left(\begin{array}{ccc|ccc} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

For clarity, I have drawn boxes around the two non-trivial sub-matrices. I repeat the point (made earlier in §2) that the insides of a matrix is generally of no practical concern to designers; the purpose of showing it explicitly here is to provide a concrete example (which any reader can check) to convince that the approach works.

Thus we have shown that the modey $\boxed{\texttt{MRC}}$ buton matrix can be constructed as a matrix, and it is therefore a linear operator — everything the paper says about matrices applies.

8.2 Modes and partial theorems

A pure modeless system might be defined as one with no partial theorems. The previous section (§8.1) constructed a matrix to handle the mode-dependent behaviour of the $\boxed{\texttt{MRC}}$ button; given this more thorough treatment of the behaviour of $\boxed{\texttt{MRC}}$, it is now easy to establish (by straight forward calculation) various user interface theorems about its action. This section reviews some theorems specific to the $\boxed{\texttt{MRC}}$ button (which the reader of this paper may easily check), but the ideas are generalisable.

It is not surprising for some matrix M that $M \neq MM$ but that $M^i = M^j$ for some small $i \neq j$. For example, as we can establish by routine calculation, $\boxed{\texttt{MRC}} \neq \boxed{\texttt{MRC}}\boxed{\texttt{MRC}}$ but $\boxed{\texttt{MRC}}\boxed{\texttt{MRC}} = \boxed{\texttt{MRC}}\boxed{\texttt{MRC}}\boxed{\texttt{MRC}}$. Thus we can prove

$$\boxed{\texttt{MRC}}^n = \boxed{\texttt{MRC}}^2 \text{ for all } n \geq 2$$

a theorem that implies that a user pressing $\boxed{\texttt{MRC}}$ more than twice is achieving nothing further, except being more certain that they have pressed $\boxed{\texttt{MRC}}$ at least twice. Note that if the $\boxed{\texttt{MRC}}$ is unreliable (e.g., the key has got some dirt in it), and the user therefore *requires* this assurance, they are on very uncertain ground since $\boxed{\texttt{MRC}}^1 \neq \boxed{\texttt{MRC}}^2$.

A pair of interesting theorems, also easily checked by calculation, are that

$$\boxed{\texttt{AC}}\boxed{\texttt{MRC}}\boxed{\texttt{MRC}} \neq \boxed{\texttt{MRC}}\boxed{\texttt{MRC}}\boxed{\texttt{AC}}$$

and

$$\boxed{\texttt{AC}}\boxed{\texttt{MRC}}\boxed{\texttt{MRC}}\boxed{\texttt{AC}} = \boxed{\texttt{MRC}}\boxed{\texttt{MRC}}\boxed{\texttt{AC}}$$

Thus, a user intent on zeroing the display and memory must do $\boxed{\texttt{AC}}$ *last*, whether or not they do it first. So although it is called 'all clear,' the user must remember

to do a memory clear with [MRC] *before* [AC]. This restriction potentially creates confusion — [AC] clears the display and [MRC] pressed twice clears the memory, but to do *both* the user has to do them in the right order. Perhaps this explains why users press buttons repeatedly to 'make sure.'

We could define matrices for the digit actions [1], [2] and so on, for the user pressing digit keys (see the web site for details). We might naïvely expect for any digit n that [n][MRC] = [MRC] but this is not so as can be seen by premultiplying both sides by [MRC], thus: [MRC][n][MRC] ≠ [MRC][MRC]. The meaning of the right hand side is now obviously that of a double [MRC] press, whereas the left hand side ends with a single press of [MRC], so it is not surprising they are unequal — they must result in different modes. This is an example of a 'partial theorem' (§4.2), partial over actions rather than states, because any action A other than [MRC] makes it true that [A][n][MRC] = [A][MRC]. More formally

$$\forall \boxed{\texttt{A}} \neq \boxed{\texttt{MRC}}, n \in \{0..9\}: \quad \boxed{\texttt{A}}\boxed{\texttt{n}}\boxed{\texttt{MRC}} = \boxed{\texttt{A}}\boxed{\texttt{MRC}}$$

A corollary of this theorem is that [n] can be a sequence of digits, not just a single digit. In fact there are many such theorems. Some certainly represent facts about the sort of behaviour that is likely to be confusing, because they are simple theorems (rules, equalities, involving only a few key presses) that are true most, but not all cases (specifically, the probably rare cases when the user presses [MRC] multiple times in succession). If a user generalises 'most of the time' to 'all of the time' they will be wrong. Since the theorems represent possible confusing behaviour, the design of the device should be appraised: can the design be changed to make the theorems general, or are the exceptions to the theorems necessary for some tasks the device should support? The partial theorems also beg empirical questions: in practice, for a realistic balance of tasks, do users worry enough about the possibly rare consequences to make a design change worthwhile?

8.3 Finding matrices from specifications

Computer algebra systems [11] are very powerful aids for doing mathematics. The purpose of this brief section is to show what can be done, using one such system, *Mathematica* [34], for concreteness. The code shown in below works fully as described, and shows how we can go from a system implementation to a matrix easily.

One way to implement an interactive system is to define each user action as a function. In a computer algebra system like *Mathematica*, we can combine doing algebra with programming functions directly. For example, a basic calculator with a memory (and a display for showing the user the results of their calculations!) may have a button [AC], and we could write code such as

```
AC[{display_, memory_, stuff__}] := {0, memory, stuff};
```

defining how `AC` operates on the state space: here, it simply sets the display to zero. The 'stuff' are other components of the state space, such as whether the calculator is on or off, what outstanding operators are (so [=] knows what outstanding operation to perform), and so on. We could have state components representing more user-oriented factors, such as the time users take. For example, we know the position of every button and so we could use Fitt's Law to estimate minimum times

for the user to be able to move from pressing one button to the next: we could have a component for time, and a component representing the co-ordinates of the last button pressed:

```
AC[{display_, memory_, time_, coords_, stuff__}] :=
{0, memory, time+moveTime[coords,{10,20}],{10,20},stuff};
```

... which supposes the AC button is at position $10, 20$. Notice how the state has now recorded the position of the user's last press, namely at $10, 20$, which in turn will be used for the timing calculation for the next button press.

Definitions like these are sufficient to prototype interactive user interfaces for on-screen simulations, and, as *Mathematica* provides a full programming language, they can implement everything any device can do. The devices do not have to be simple pushbutton type devices: it would be possible, say, to implement the automatic suspension and enhanced braking systems of an advanced car and how they interact with the user and driving conditions.

One way to obtain matrices from such definitions is to introduce a function `proj` that projects the full state space to vector spaces that represent the features of particular interest. We then ask *Mathematica* to find the matrices (if they exist) that correspond to the operations such as AC. If the matrices do not exist, then we have over-simplified in the projection or the generality of our assumptions. In practice it is easy to find out what the problems are, but in this paper I shall ignore these technical details.

For example, we can define `proj` to extract just the display and memory contents from the state space:

```
proj[{display_, memory_, _}] := {display, memory};
d = 6; example dimensions of full implementation state space
v = Array[Unique[], {d}]; symbolic state vector
d = 2; example dimensions of projected state
M = Array[Unique[], {d,d}]; symbolic matrix
M /. First@SolveAlways[proj[AC[v]] == proj[v].M, v]
```

Although the example here, to find the matrix corresponding to `AC`, is trivial (so that I can show the full and complete code to solve the problem), a lot of work can be done by the built-in function `SolveAlways`, which here is finding the matrix M such that $\texttt{proj}(\texttt{AC}(v)) = \texttt{proj}(v)M$ for any v. With the definition of `AC` given above, this code obtains the matrix

$$\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$$

— which is one of the matrices used in §6. The code above was an illustration of how an arbitrary function (here, `AC`) acting on a 6 dimensional state space could be converted to a 2×2 matrix. We don't always want to do this! As a general purpose programming language, *Mathematica* allows the derivation to be generalised: we can make `AC` a parameter, so we can find the matrix for any function, and we can use nicer notation. We can define $\langle f \rangle$ to denote the matrix in the projected space corresponding to any function f implementing a user interface action. We can now directly establish homomorphisms such as

$$\langle f \circ g \rangle = \langle g \rangle \cdot \langle f \rangle$$

and we can use this notation to try them out in *Mathematica*.[17] Here, if f was the function implementing a calculator's 'add to memory' button and g the function that implements the 'subtract from memory' button, the homomorphism here shows that the matrix corresponding to the functional composition of the functions for the buttons M+ and M− is equal to the product of the two matrices corresponding to each button individually. If this equation is typed into *Mathematica* (after appropriate definition of the button functions), the answer `True` will be immediately obtained. We might establish many laws of this form to convince ourselves that all matrices we are using to analyse a user interface are correctly modelling some system we have implemented.

I gave examples like this in the Section 6, where simple 2×2 matrices were used. With *Mathematica* we could easily do similar calculations with possibly huge matrices, derived reliably from real, working system definitions.

8.4 Design tools

Addressing the problems designers may have in using matrices in design is a matter of some urgency. This paper showed that established, 'simple' interactive systems that we all take for granted have non-trivial design problems. Whether these design problems are always or sometimes *actual* problems for users — whether frequent, critical, trivial or merely irritating — is another matter, but it seems plausible that for many safety critical devices, such as electronic medical equipment and avionics systems it would be wise to assume that avoidable design problems should be avoided. Currently, HCI professionals are not addressing these issues, and if interactive safety critical systems are not being designed to any higher standards, we should worry for the HCI consequences. In many areas, I would argue that designers (or at least some people on the design teams) have an obligation to learn matrix algebra or an equivalent formalism to ensure that the systems they build really do what they intend. Too often (in my opinion) the 'HCI design' team's job has been seen as instructing the programmer and other technical designers: these people need to be involved in the HCI design, just like all the other disciplines such as anthropology.

Modelling techniques can be made available to designers, fully automated or partly automated: one can imagine automatic design tools that help designers do with ease some of the sorts of analysis that in this paper have been spelt out explicitly and at length. Design tools can be used to generate accurate system models for psychological modelling, and they would help reduce the gap between psychological analysis and what a human-machine system actually does. Some initial progress has already been achieved building matrix-based modelling tools [12].

See [26] for a wide range of design benefits from a matrix-based approach, in particular showing concretely how a device can be simulated, analysed and have

[17] Details for programmers: The built-in function composition operator is defined as ∘, and the equals operator is normally typed as `==` (to distinguish it from assignment, which is a single `=` symbol). The size of the matrix and the choice of projection function can be defined in the usual way by using `Assuming[]`.

various outline user manuals generated automatically and reliably.

8.5 Constructive use of matrices

The examples in this paper are all based on reverse engineering: if something is already designed — indeed, an already working product — I can evidently present its matrices, which might appear to be more-or-less pulling out of thin air. But how useful are matrices for a *constructive* design process for ordinary designers, who perhaps are not as interested in matrices as I am? Imagine a designer building a system: how would matrices help during, rather than after building it? What properties should be explored over unfinished designs?

In an unfinished design, we have to distinguish between partial theorems and theorems that are tentatively partial because the design is not complete. As we make forward design decisions, the design is not complete. This is a stark contrast to the fully-informed hindsight that reverse engineering affords. Whilst the hindsight of this paper could improve iterative design (e.g., if the manufacturers produce new models based on the existing models), it is not so clear how it contributes to innovation when less information is available.

It is not obvious that matrices can easily define all systems of interest, so any athe research and development programme that developed a tool could anticipate meeting a future barrier. It is possible that a designer must have some feature (or feature interaction) that the proposed tool cannot support. This is a risk. In contrast, if the design notation was general purpose (e.g., a programming language like Java) rather than matrix algebra, then the research investment would obviously be worthwhile: there are no likely barriers to progress — and certainly no self-imposed barriers that competitors could avoid. The suggestion of §8.3 is that some general purpose languages could support a matrix based design approach automaticaly, but, even so, building tools for them would be a non-trivial task (and probably pointless) because it requires implementing powerful computer algebra engines.

To summarise: developing a good design tool for matrices will require considerable commitment (e.g., motivated by a safety critical design problem) and will have to be done under assumptions of uncertain payoffs in the long run. In contrast, and competing with such investment of effort, there are many Rapid Application Development (RAD) environments for general purpose systems such as Java.

8.6 Model checking and other approaches

Model checking and theorem proving are standard techniques for establishing that system specifications have desired properties. Typically they require expertise beyond basic matrix algebra, but the tradeoffs are very different — both approaches are very well supported with sophisticated (though specialised) tools. Useful references to follow are [8] for model checking, and [9] for theorem proving applications in HCI contexts. Unlike the simple matrix algebra approach, promoted by this paper, making effective use of either model checking or theorem proving requires learning the particular notations used by tools that support the approaches, as well as having a sophisticated understanding of the underlying mathematical logics.

8.7 Modes and the ordering states

The text of the paper explained how partitioned matrices can be used to handle modes and subsystems well. We gave the impression that any system can be handled by partitioning. A partition is a partial ordering of states; for example, if a system has two modes $\mathcal{A}$ and $\mathcal{B}$, then we could partition the button matrices on the assumption that all states in mode $\mathcal{A}$ are numbered less than all states in mode $\mathcal{B}$. Then we can easily write down a partitioned matrix. A different pair of modes, say $\mathcal{B}$ and $\mathcal{C}$, may impose a different partial ordering, and in principle it is possible for there to be no consistent total ordering of states for all three modes. At first sight, this problem does not matter, since if M is a partitioned matrix separating modes $\mathcal{A}$ and $\mathcal{B}$, we might use a matrix N such that $M = PNP^{\mathsf{T}}$ to deal with modes $\mathcal{B}$ and $\mathcal{C}$. (See §7.3.)

We can thus handle 'inconsistent' modes with more than one matrix (in this example, M and N, but $M \neq N$ even though they represent the same button matrix). Is it ever the case that a user needs to understand all inconsistent modes at once? If so, there can be no single partitioned matrix that represents this situation. A counter argument is that if there is no such matrix, then we can claim the system is badly designed. If it really is the case that such a system is 'bad' (for some senses of 'bad'), then a design notation that avoids such problems is to be welcomed. On the other hand, it is possible that some tasks require inconsistent modes for good usability reasons — and as this is a case that would be awkward to handle using matrices we might take it as an argument against matrix algebra in user interface design.

I know of no such examples, but finding any would either help sharpen the boundaries of matrix algebra as applied to user interface design, or perhaps would completely undermine the whole enterprise.

8.8 Complexity theory

The set of button matrices defining a system are (in an important sense) a complete interaction definition. The matrices often have structure that can be exploited to make more compact definitions. In general, a set of matrices can be compressed depending on the Kolmogorov complexity, which is therefore a measure of the formal complexity of the interactive system. If a user is to learn how to use a system in detail they have to know the matrices or equivalent (and anything equivalent has the same Kolmogorov complexity), so this complexity is a theoretical low bound on the cognitive resources the user needs: how hard this knowledge is to acquire, remember and recall for use.

Kolmogorov complexity is not the only measure of complexity. Fisher complexity is based on a notion of observation: this might be relevant here, as we are concerned with the complexity of the user model given that users observe system transitions and their effects. Finding a complexity metric that is well defined and has some or all of the appropriate cognitive properties would be a major but worthwhile research project.

8.9 Pseudo inverses

If a matrix is singular it has no inverse, but this is a strict criterion that may have little relevance to a user (see §4.2): if a user can undo an operation almost all of

the time, rather than all of the time, this may be sufficient. For example, a button may have an effective inverse except when the device is off — which means a user would probably not have tried the action that had no inverse!

A more realistic criterion if A is singular is to find a matrix B such that $AB \approx I$ within some specified accuracy, perhaps weighted with known or guessed probabilities of users having the device in certain states. Of all the matrices B that satisfy this, to find one that is 'most' relevant to users is a challenge: for example the easier B is to factorise in terms of existing button matrices the easier a user will be able to undo the action $\boxed{\mathtt{A}}$.

There are standard generalised inverses, which exist even when a matrix is singular. For example, the Moore-Penrose pseudo inverse of A, written $A^{(-1)}$, satisfies $AA^{(-1)}A = A$. (In the special case that A is invertible, then $A^{(-1)} = A^{-1}$.) This equation has various user interface design readings, for instance: doing any sequence of actions that are equivalent to $\boxed{\mathtt{A}}^{(-1)}$ after first doing $\boxed{\mathtt{A}}$ can be undone by doing $\boxed{\mathtt{A}}$ again; or it can be read as: if a user does $\boxed{\mathtt{A}}$ and changes their mind (perhaps because $\boxed{\mathtt{A}}$ was thought to be a slip), they can do $\boxed{\mathtt{B}}$, and if they change their minds back (perhaps because doing $\boxed{\mathtt{A}}$ was not a slip — doing $\boxed{\mathtt{A}}$ was in fact correctly intended) then they need only do $\boxed{\mathtt{A}}$ again. The Moore-Penrose pseudo inverse is defined to minimise the sum of the squares of all 'errors' in $AA^{(-1)} - I$, so this is a form of partial theorem, $AA^{(-1)} \approx I$.

The pseudo inverse suggests new avenues of research, including the following:

—Given a partial theorem, find a design (from a matrix) that minimises some measure (in the Moore-Penrose, minimise the sum squared error). What measures are effective?

—How should the measure be weighted from empirical data of user behaviour?

—The partial theorem here, namely interpretations of $ABA = A$, is clearly one of a class of interesting theorems, some of which may more closely fit use scenarios, particularly in representing forms of error recovery. Defining an effective repertoire of theorems, together with the conditions under which they are applicable, would be very fruitful.

—and what other concepts from matrix algebra can be recruited to address user interface design issues?

9. CONCLUSIONS

Standard matrix algebra gives detailed and specific insight into user interface design issues. Our use of matrix algebra in this paper reveals persuasively that some apparently trivial user interfaces are very complex. I suggest that the arbitrary complexity we uncovered in some systems is a sign of bad design (it certainly is bad engineering, if not bad user interface design), and that the arbitrary complexity begs usability questions that should be addressed by employing empirical evaluation methods.

This paper raises a question:

If some user interface design issues are so complex they are only understandable with the help of algebra (or other mathematical approaches), how can ordinary usability evaluation methods (which ignore such formalisms) reliably help?

In areas of safety critical systems, this is a very serious question. At least one response is to design systems formally to be simpler, so that there are plausibly no such overwhelmingly complex usability issues to find, fix or palliate. Another response is to use formal approaches, such as matrix algebra, to support design so that designers know what they implementing: the results of usability evaluations can then be used to help fix rather than disguise problems. Even where formal methods appear not to be applicable, at least the designers would know, and they would then take extra precautions, whether in their design processes or in the design of the user interface features.

Reasoning about user interfaces at the required level of detail is usually very tedious and error prone — and is usually avoided! Matrices enable calculations to be made easily and reliably, and indeed also support proofs, of both general and detailed user interface behaviour.

Very likely, as suggested in the box above, the extreme complexity of doing some tasks (such as stoing to memory) are why users, designers and usability professionals have all but ignored the poor usability of some devices. They are far too hard to think about! The intricacy and specificity of feature interactions (as in the digital multimeter) apparently encourage problems to be overlooked, or, more likely, never to be discovered.

For the time being it remains an unanswered empirical question whether the apparently unnecessary complexities of the Casio calculator or the Fluke 185 make them superior designs than ones that we might have reached driven by the æsthetics of matrix algebra. However, it is surely an advantage of an algebraic approach that known (and perhaps rare) interaction problems can be eliminated at the design stage, and doing so will strengthen the validity of any further insights gained from employing empirical methods. Certainly, in comparision with informal evaluations (e.g., [25], which comments on other Fluke instruments), a matrix algebra approach can provide *specific* insights into potential design improvement, and ones moreover that are readily implemented. There is much scope for usability studies of better user interfaces of well-defined systems, rather than (as often happens) studies of how users cope with complex and poorly defined interfaces.

Many centuries of mathematics have refined its tools for effective and reliable human use — matrix algebra is just one example — and this is a massive resource that user interface designers should draw on to the full.

Acknowledgements

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

[1] M. A. Addison & H. Thimbleby, "Intelligent Adaptive Assistance and Its Automatic Generation," *Interacting with Computers*, **8**(1), pp51–68, 1996.

[2] J. L. Alty, "The Application of Path Algebras to Interactive Dialogue Design," *Behaviour and Information Technology*, **3**(2), pp119–132, 1984.

[3] J. R. Anderson & C. Lebiere, *The Atomic Components of Thought*, Lawrence Erlbaum Associates, 1998.

[4] A. Blandford & R. M. Young, "Specifying User Knowledge for the Design of Interactive Systems," *Software Engineering Journal*, **11**(6), pp323–333, 1996.

[5] J. R. Brown, *Philosophy of Mathematics*, Routledge, 1999.

[6] C. G. Broyden, *Basic Matrices*, Macmillan, 1975.

[7] S. K. Card, T. P. Moran, & A. Newell, *The Psychology of Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1983.

[8] E. M. Clarke, Jr., O. Grumberg & D. A. Peled, *Model Checking*, MIT Press, 1999.

[9] P. Curzon & A. Blandford, "Detecting Multiple Classes of User Errors," M. Reed Little & L. Nigay, editors, *Engineering for Human-Computer Interaction*, LNCS **2254**, pp57–71, Springer Verlag, 2001.

[10] A. Dix, J. Finlay, G. Abowd & R. Beale, *Human-Computer Interaction*, 2nd. ed., Prentice Hall, 1998.

[11] J. von zur Gathen & J. Gerhard, *Modern Computer Algebra*, Cambridge University Press, 2003.

[12] J. Gow & H. Thimbleby, "MAUI: An Interface Design Tool Based on Matrix Algebra," *ACM Conference on Computer Aided Design of User Interfaces*, CADUI **IV**, edited by R. J. K. Jacob, Q. Limbourg & J. Vanderdonckt, pp81–94 (pre-proceedings page numbers), 2004.

[13] H. R. Hartson, A. Siochi & D. Hix, "The UAN: A User Oriented Representation for Direct Manipulation Interface Designs," *ACM Transactions on Information Systems*, **8**(3), pp181–203, 1990.

[14] I. Horrocks, *Constructing the User Interface with Statecharts*, Addison-Wesley, 1999.

[15] M. Y. Ivory & M. A. Hearst, "The State of the Art in Automating Usability Evaluation of User Interfaces," *ACM Computing Surveys*, **33**(4), pp470–516, 2001.

[16] S. C. Johnson, "Yacc: Yet Another Compiler-Compiler," *UNIX Programmer's Manual*, **2**, Holt, Rinehart, and Winston, New York, NY, pp353–387, 1979.

[17] D. Kieras & P.G. Polson, "An Approach to the Formal Analysis of User Complexity," *International Journal of Man-Machine Studies*, **22**(4), pp365–394, 1985.

[18] D. E. Knuth, "Two Notes on Notation," *American Mathematical Monthly*, **99**(5), pp403–422, 1992.

[19] L. Lamport, "TLA in Pictures," *IEEE Transactions on Software Engineering*, **21**(9), pp768–775, 1995.

[20] B. Myers, "Past, Present, and Future of User Interface Software Tools," in J. M. Carroll, editor, *Human-Computer Interaction in the New Millenium*, Addison-Wesley, 2002.

[21] W. M. Newman, "A System for Interactive Graphical Programming," *Proceedings 1968 Spring Joint Computer Conference*, 47–54, American Federation of Information Processing Societies, 1969.

[22] D. L. Parnas, "On the Use of Transition Diagrams in the Design of a User Interface for an Interactive Computer System," *Proceedings 24th. ACM National Conference*, pp379–385, 1964.

[23] S. J. Payne, "Display-Based Action at the User Interface," *International Journal of Man-Machine Studies*, **35**(3), pp275–289, 1991.

[24] P. Pirolli & S. K. Card, "Information Foraging Models of Browsers for Very Large Document Spaces," ACM Proceedings Advanced Visual Interfaces, AVI'98, 83–93, 1998.

[25] J. Raskin, *The Humane Interface*, Addison-Wesley, 2000.

[26] H. Thimbleby, "Specification-led Design for Interface Simulation, Collecting Use-data, Interactive Help, Writing Manuals, Analysis, Comparing Alternative Designs, etc," *Personal Technologies*, **4**(2), pp241–254, 1999.

[27] H. Thimbleby, "Calculators are Needlessly Bad," *International Journal of Human-Computer Studies*, **52**(6), pp1031–1069, 2000.

[28] H. Thimbleby, P. Cairns & M. Jones, "Usability Analysis with Markov Models, *ACM Transactions on Computer Human Interaction*, **8**(2), pp99–132, 2001.

[29] H. Thimbleby, "Analysis and Simulation of User Interfaces," *Human Computer Interaction 2000*, BCS Conference on Human-Computer Interaction, edited by S. McDonald, Y. Waern and G. Cockton, **XIV**, pp221–237, 2000.

[30] H. Thimbleby, "Permissive User Interfaces," *International Journal of Human-Computer Studies*, **54**(3), pp333–350, 2001.

[31] H. Thimbleby, "Reflections on Symmetry," *Proc. Advanced Visual Interfaces*, AVI2002, pp28–33, 2002.

[32] H. Thimbleby & J. Gow, "Computer Algebra in Interface Design Research," *2004 ACM/SIGCHI International Conference on Intelligent User Interfaces*, IUI 04, edited by N. J. Nunes & C. Rich, pp366–367, 2004.

[33] A. I. Wasserman, "Extending State Transition Diagrams for the Specification of Human Computer Interaction," *IEEE Transactions on Software Engineering*, **SE-11**(8), pp699–713, 1985.

[34] S. Wolfram, *The Mathematica Book*, 4th. ed., Cambridge University Press, 1999.

[35] R. M. Young, "Surrogates and Mappings: Two Kinds of Conceptual Models for Interactive Devices," D. Gentner & A. L. Stevens (eds.), *Mental models*, pp35–52, Hillsdale, NJ: Lawrence Erlbaum Assoc, 1983.

[36] R. M. Young, T. R. G. Green & T. Simon, "Programmable User Models for Predictive Evaluation of Interface Designs," ACM Proceedings CHI'89, pp. 15–19, 1989.

APPENDIX

A. SIMPLE MATRIX ALGEBRA

Matrices are rectangular arrays of *elements*, usually numbers (but see §A.1), that can be added and multiplied. If M is a matrix, then m_{ij} is the conventional way to refer to the element of M on row i and column j. A matrix with the same number of rows as columns is a *square* matrix.

Matrices are usually shown between round brackets, hence

$$M = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix}$$

for a two-by-two matrix. Matrices need not be square, but they must be rectangular, and they can be of any size or *dimension*, from 1×1 up to infinity. One of the advantages of using matrices is that a single concise letter, such as M, can refer to the thousands or millions of numbers that are the elements of M. This is a great aid to comprehensibility.

Matrices are equal when they are the same size and all their corresponding elements are equal.

Matrices are added together by adding their corresponding elements, but multiplication is more interesting. To calculate a matrix product $AB = C$ of two matrices A and B, each element of row i of A is multiplied by each element of column j of B. The individual products are added together to make element c_{ij}. For small two-by-two matrices, the calculation is not too onerous:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{21} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{21} + a_{22}b_{22} \end{pmatrix}$$

Thus the product of two matrices A and B, written AB, is very convenient notation for a complex operation.

Square matrices can be raised to powers: M^n means the matrix M multiplied by itself n times. Thus $M^1 = M$, $M^2 = MM$, $M^3 = MMM$ and so on.

The special case of a matrix with one row or one column is called a *vector*. A vector $\mathbf{v}$ is conventionally written in bold or (especially when written by hand) with a bar, as in $\underline{v}$. Multiplying vectors and matrices follows the same rules: rows get multiplied point-wise by columns and added together. In this paper, all vectors happen to be row vectors. The multiplication works as follows:

$$\mathbf{v}M = (v_1\ v_2) \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} = (v_1m_{11} + v_2m_{21} \quad v_1m_{12} + v_2m_{22})$$

Notice that all four elements of the matrix M are required to specify the two elements of the product $\mathbf{v}M$. From the rules of multiplication, it follows that multiplying a vector by a matrix (in that order) requires a row vector, whereas multiplying a matrix by a vector (in that order) requires a column vector.

The *identity matrix* is usually written I, and is a square matrix that consists of zero elements except on the leading diagonal, where all elements are 1. The 3×3 identity matrix looks like this:

$$I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Any matrix or vector multiplied by the identity matrix (of appropriate size) is unchanged. Although matrices can be any size, we usually do not bother to specify what size the identity matrix is: it has to be square, and it has to be the same size as the matrices it is being used with — so its size is always implicit in the equations where it is used. Any square matrix raised to power zero is the identity matrix (of the same size): thus for any M, $M^0 = I$.

The *zero* matrix $\mathbf{0}$ is a matrix with all elements zero. The zero matrix behaves like zero in other algebraic systems: $\mathbf{0}M = \mathbf{0}$ (multiplying by zero is zero), and $\mathbf{0} + M = M$ (adding zero leaves unchanged).

The *inverse* of a matrix M is a matrix written M^{-1} and is such that M times M^{-1} is the identity matrix; or in formula form: $MM^{-1} = I$. It is easy to show that the inverse of a matrix is unique, but not all matrices have inverses. A matrix with no inverse is *singular*.

There is a special function of matrices, called the determinant, which has the property that the determinant of a matrix is zero exactly when the matrix has no inverse. So if $\det A = 0$ (the determinant is also written $\det A = |A|$) then there is no matrix $B = A^{-1}$ such that $AB = I$ or $BA = I$.

In general, it is quite tricky to calculate the inverse of a matrix, even if there is one. See §4.1 for a discussion of the relevance of inverses to button matrices.

The *transpose* of a matrix M is written M^T and has the rows and columns of M

swapped. Thus

$$M^{\mathsf{T}} = \begin{pmatrix} m_{11} & m_{21} \\ m_{12} & m_{22} \end{pmatrix}$$

which swaps the off-diagonal elements m_{12} and m_{21}. The elements on the diagonal are not changed when a matrix is transposed.

A.1 Partitions

Matrices are normally considered as arrays of numbers; in fact any values can be used, provided they can be 'added' and 'multiplied.' Matrices themselves have these properties, and therefore matrices can be built up, or partitioned, out of elements that are themselves matrices, or *submatrices*. Consider a matrix M *partitioned* into four submatrices:

$$M = \left(\begin{array}{c:c} M_{11} & M_{12} \\ \hdashline M_{21} & M_{22} \end{array}\right)$$

Partitioned matrices are sometimes called *block* matrices, as their elements form blocks. In this paper, I use dotted lines like ····· to visually clarify the structure of a partitioned matrix. The dotted lines have no mathematical significance.

The submatrices, M_{11} *etc*, need not all be the same size, though they must conform (i.e., be the right size) for the operations required of them. Thus if this partitioning of M is used to multiply a state vector $\mathbf{s}$, partitioned as $(\mathbf{s}_1 \vdots \mathbf{s}_2)$, then the number of columns of $\mathbf{s}_1$ must equal the number of rows of M_{11} *etc*. If $\mathbf{s}$ and M conform in this way, the partitions can be multiplied out:

$$\left(\begin{array}{c:c} \mathbf{s}_1 & \mathbf{s}_2 \end{array}\right) \left(\begin{array}{c:c} M_{11} & M_{12} \\ \hdashline M_{21} & M_{22} \end{array}\right) = \left(\begin{array}{c:c} \mathbf{s}_1 M_{11} + \mathbf{s}_2 M_{21} & \mathbf{s}_1 M_{12} + \mathbf{s}_2 M_{22} \end{array}\right)$$

Mathematically this is necessarily correct, regardless of the structure of the FSM and buttons we might be describing.

We can see, for example, that if M_{21} is zero, the complete behaviour of the $\mathbf{s}_1$ component of the system can be understood even ignoring the $\mathbf{s}_2$ component:

$$\left(\begin{array}{c:c} \mathbf{s}_1 & \mathbf{s}_2 \end{array}\right) \left(\begin{array}{c:c} M_{11} & M_{12} \\ \hdashline \mathbf{0} & M_{22} \end{array}\right) = \left(\begin{array}{c:c} \mathbf{s}_1 M_{11} & \mathbf{s}_1 M_{12} + \mathbf{s}_2 M_{22} \end{array}\right)$$

Whether a user fully understands $\mathbf{s}_1$ or not (they may understand a subpartition of it), no knowledge of $\mathbf{s}_2$ need interfere with their understanding. Whatever befalls the $\mathbf{s}_2$ component, we can abstract the properties of $\mathbf{s}_1$ and M_{11} independently. Similarly we can understand the $\mathbf{s}_2$ component of the system completely even ignoring the $\mathbf{s}_1$ component provided M_{12} is zero, and so on. Since it seems desirable to design user interfaces where users can understand parts of the system independently, designing systems with zero submatrices is a powerful design heuristic. In fact, it is not just a design heuristic but a heuristic for calculation: if a

matrix can be factored into partitioned matrices with 'big enough' zero blocks, multiplication, as a calculation, can be speeded up. Whether this means that button matrices with large zero blocks are 'easier to think about' is an interesting question for investigation.

A.2 Algebraic properties

Algebra looks at the properties of matrices, rather than how calculations are done with them — this is why algebra is an appropriate approach to user interaction: we are interested in the properties of interaction, which matrix algebra defines. We are not interested in *how* the user thinks, but we have a convenient way to calculate with matrices (which is not available to the typical user).

Here are a few basic algebraic properties:

—Any matrix M times the identity is itself: $MI = IM = I$.

—The inverse of a matrix M is written M^{-1}, and $MM^{-1} = M^{-1}M = I$. The inverse of a matrix is unique. Not all matrices have inverses, however. Inverses are closely related to undo; see §4.1.

—Matrix multiplication does not commute. In general, $AB \neq BA$, although for some matrices $AB = BA$. For any matrix M, $MI = IM$.

—Matrix multiplication is associative: $(AB)C = A(BC)$, hence we can write ABC without ambiguity. Associativity is closely related to modelessness; see §6.

—If a matrix has numerical elements, its determinant is a number. If the determinant of M is zero, $\det M = 0$, then M is singular.

—The determinant of a matrix product is the product of the determinants: $\det AB = \det A \det B$.

—Matrices do not support division; $\frac{A}{B}$ makes no sense, since in general $AB^{-1} \neq B^{-1}A$.

—Transposition reverses the order of multiplication: $(AB)^\mathsf{T} = B^\mathsf{T}A^\mathsf{T}$. Other laws for transposition include $I^\mathsf{T} = I$, and for any matrix $M^{\mathsf{T}\mathsf{T}} = M$.

B. 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 [21] and Parnas [22] in the 1960s, and reaching a height of interest in user interface management systems (UIMS) work [33]; see [10] for a textbook introduction with applications of FSMs in HCI. FSMs can handle paralleism, nondeterminism, and so forth (many parallel reactive programming languages compile into FSM implementations).

Now the concerns of HCI have moved on [20], beyond any explicit concern with FSMs — a continual, technology-driven pressure, but one that tends to leave open unfinished theoretical business.

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 paper, it is convenient to use button names for matrices. In particular this saves us inventing mathematical names for symbolic buttons, such as ▽. (See Appendix C for a formal perspective on this convenience.)

The state of the FSM is 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. The 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 they as ordinary users 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 factorisation of M as a product of matrices B_1, B_2, etc, that are available as actions to them, such as $M = B_1B_2$, meaning 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.

B.1 Aren't FSMs too restricted?

Many elementary references in HCI dismiss FSMs (e.g., [10]). Since button algebras are formally isomorphic to FSMs it is worth exploring some of the issues:

FSMs are large. FSMs for typical devices often have thousands of states, if not

more. The size of a FSM is not a concern for this paper, though obviously it would be a serious problem if one wanted to draw the corresponding transition diagram (though [19] provides a solution). No finite state machines have been or need to be drawn for this paper (though I drew two in the Figures for purely explanatory purposes).

FSMs are unstructured. FSMs are indeed unstructured. Matrices, however, can be partitioned; Example 3 in this paper and the handling of the MRC button (in §mrcap) shows how a hierarchical definition can be constructed using matrices, and hence how a very large system can be modelled.

FSMs are finite. FSMs are finite and therefore formally less powerful than infinite computational models such as push down automata (PDAs). A calculator using brackets is readily modelled as a PDA, and therefore one might think it is not a FSM. However all physically realisable digital devices are FSMs, whether or not it is convenient to model them explicitly as such. The matrix approach can be used to model finite PDAs.

FSMs are not relevant to users. Certainly, FSMs are mathematical structures and they do not exist in any concrete form for users. Users need not be expected to reason about the behaviour of FSMs: they are typically far too big, and as a formalism they are so versatile that they have no structure that really helps thinking about them. Obviously some FSMs will have interesting structure (e.g., ones designed using Statecharts [14]), but in these cases it is easier to think about the structure than the FSM itself. What the user can see, however, is buttons and their effects. This paper shows that each button is a matrix; it thus turns out that users — whether they know it or not — are doing matrix algebra when they press buttons.

Systems are not implemented as FSMs. Most systems are implemented in *ad hoc* ways, and determining any model from them is hard if not impossible. In this sense, FSMs suffer from problems no different from any other formal approach. Better, one would start with the formal model and derive (preferably automatically) the implementation.

FSMs and their button matrices are sparse. There are many techniques for representing them in a compressed form, and many of these techniques (e.g., black box linear algebra [11]) are equivalent to conventional program implementation techniques where the FSM structure is not explicit.

FSM models are impossible to determine. On the contrary, if systems are developed rigorously it is not hard to determine finite models of user interfaces from them: it is a routine application of a computer algebra system — see §8.3 for further details.

Overall, there is a balance to be had. It may not be possible to represent all aspects of a system in a clear way using matrix algebra. But representing how components of state are manipulated by the user can nevertheless reveal complications in the way the device works, complications that may perhaps cause problems for users, and will certainly highlight issues designers should consider. Indeed, we can talk about user actions and their algebra without referring to state, and thus, at least algebraically, we can hide the state size issues. Further arguments can be found in [28].

C. FORMAL DEFINITIONS AND BASIC PROPERTIES

It perhaps helps some readers to give at least one formal definition — there are many ways and styles to formalise the approach.

A FSM is a tuple $F = \langle S, \Sigma, S_0, \delta \rangle$, where S is a set of states, Σ an alphabet (e.g., actions, buttons), $S_0 \subseteq S$ the set of initial states, $\delta \subseteq S \times \Sigma \times S$ the transition relation. The definition is standard.

This is a mathematical definition; we are not (in this Appendix) concerned with what F is or how it is realised, only with its formal properties. We may think of F as a box with buttons and electronics in it, but it could equally be a human, a robot, or a horse, or any finite discrete system where we are interested in its actions and their laws.

For some issues of human computer interaction, we might want to introduce refinements to the FSM definition, such as representations for states. For example, the user can see an image of the state $i(s)$ on most systems (on simple systems i may be a bijection, otherwise users cannot be certain of the state).

Choose $n : 1..|S| \leftrightarrow S$, a bijection, taking numbers to states.

Let $\mathcal{B} : \Sigma \to M$ give the button matrix an action denotes. Specifically, for the action $\sigma \in \Sigma$ the button matrix $B = \mathcal{B}[\![\sigma]\!]$ over Booleans is defined by the characteristic function of the transition relation:

$$b_{ij} = \delta(n(i), \sigma, n(j))$$

So for action σ, B is a matrix of Boolean values, with True elements when the FSM has a transition for the action.

Analogously we define an intial state vector $\mathbf{v}$:

$$\mathbf{v}_i = n(i) \in S_0$$

> THEOREM. A FSM is isomorphic to a set of button matrices $\{\mathcal{B}[\![\sigma]\!] : \sigma \in \Sigma\}$ equipped with matrix multiplication together with an initial state vector $\mathbf{v}$.

Matrices can have elements over any ring, Booleans, integers, reals, and so on. It is briefer to write 0 and 1 instead of False and True, a convention we adopt. We take the ring to be implicit from the context. In the body of the paper, explicit use of $\mathcal{B}$ was avoided; hence by convention, for a button $\boxed{\vee}$ we simply denote the corresponding button matrix $\boxed{\vee}$ instead of $\mathcal{B}[\![\boxed{\vee}]\!]$; thus avoiding typographical clutter.

Two important properties can immediately be stated.

—A *deterministic* action is one that has no more than one 1 in each row of its matrix. If a FSM is deterministic, then the initial state S_0 is a singleton (i.e., there is exactly one initial state), and the transition relation δ is a function $S \times \Sigma \to S$. The paper only considers deterministic FSMs.

—An *unguarded* action is one that has at least one 1 in each row of its matrix. Normally we are interested in deterministic, unguarded actions. A guarded action is one that cannot be attempted in some states: this is unusual, but can happen when systems have protections or guards that stop users reaching some buttons. Normally, of course, a user can try to do actions whether or not the designer

intended those actions to do anything: if the user can actually do the action, it is unguarded and its meaning in all states must therefore be defined (so there must be at least one 1 in each row of its matrix).

There are two extreme cases of button matrices. The null matrix, $\mathbf{0}$, is all zeroes and is therefore guarded: but it is guarded in every state and therefore cannot be used under any circumstances. It is a button (or lever, etc) that could be removed from the user interface without changing the functionality of the design. The identity matrix, I, has 1s down its diagonal (so for a matrix I, $I_{ii} = 1$ and $I_{ij} = 0$ for all $i \neq j$); it is not guarded but achieves no change in any state. The action can always be done by the user, but the action does not change the state. In both cases, null and identity, removing the 'useless' (merely decorative?) buttons might improve the user's performance, as the interface would be simplified.

It is trivial to check these properties. For exampl, the two matrices that can be written down directly from the diagram in Figure 1b are guarded, thus indicating either that there is an error in the system specification or that the designer has assumed the actions are guarded by some shield, safeguard, lockout or other protection. A borderline possibility is covered by *affordance* [31], in that the physical design of the system may sufficiently discourage the user from trying the action: as an example, consider a toggle switch with on and off actions. When it is in its 'on' position, it is or should be 'obvious' that it cannot be switched even further on; or instead of a toggle switch, there may be two buttons, which each press in and stay in for their actions 'on' and 'off.' Once either button has been pressed in, it isn't possible to press it in further. In this case, the actions are guarded, and some rows of the corresponding matrix will be zero.

If a button is non-deterministic, pressing it may achieve unpredictable results for the user — this is generally undesirable, and a designer should check that button matrices are deterministic. However, although we are certain what a button does, we may not be sure what the user will do: which button will they press? If p_i is the probability that action Σ_i is undertaken by a user, then $\sum p_i = 1$ (since with probability 1 the user will do something), and

$$\sum_i p_i \mathcal{B}[\![\Sigma_i]\!]$$

is a stochastic probability matrix, which can be analysed to obtain statistical information about the user or about the design of the user interface. Such statistical models are explored in detail in [28].

For 0/1 integer matrices

$$M = \max(B_1, B_2, B_3, \ldots)$$

is the conventional FSM transition matrix, which has many simply-stated usability properties. For example, M^d gives, for all pairs of states, the number of ways of reaching any state from any other in exactly d steps. The higher $(M^d)_{ij}$, then, in some sense the easier a state j is to reach from state i. In particular, if any element of $M^{|S|}$ (or any larger power of M) is zero, a user will find it *impossible* to reach some states. Unreachability is a generally undesirable user interface

property. These two examples are provided as simple illustrations of the way that formal methods can make different types of claim: either plausible or precise usability claims. In the case of a user finding some tasks 'easier' the higher some elements of M^d, this in general will depend on the particular user interface and other concrete issues. Thus this claim raises an empirically testable question for any particular human/system configuration or task scenario. On the other hand, if a system has unreachable states, all users will find certain tasks impossible regardless of specific circumstances — and knowing this requires no empirical testing.