Usability Analysis with Markov Models

HAROLD THIMBLEBY, PAUL CAIRNS, and MATT JONES
Middlesex University

How hard do users find interactive devices to use to achieve their goals, and how can we get this information early enough to influence design? We show that Markov modeling can obtain suitable measures, and we provide formulas that can be used for a large class of systems. We analyze and consider alternative designs for various real examples. We introduce a “knowledge/usability graph,” which shows the impact of even a small amount of knowledge for the user, and the extent to which designers’ knowledge may bias their views of usability. Markov models can be built into design tools, and can therefore be made very convenient for designers to utilize. One would hope that in the future, design tools would include such mathematical analysis, and no new design skills would be required to evaluate devices. A particular concern of this paper is to make the approach accessible. Complete program code and all the underlying mathematics are provided in appendices to enable others to replicate and test all results shown.

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: Markov models, usability analysis

1. INTRODUCTION

How can products be designed to be more usable? One answer to this question is that usability depends on the match between the product and the users under the particular constraints of the environment and tasks being performed with the product. The problem is that usability, seen like this, depends on the world when the product is used *not* when it is designed. So if we want to design better products, foresight has to be used. Foresight can be based on case histories from previous product evaluations; sometimes foresight can be focused by general psychological or socio-technical knowledge. In all approaches, design for usability requires considerable expertise and commitment to usability, neither of which is conventionally nor realistically available in the crucial early stages of technical design. Often one therefore seeks improvements in usability *after* the initial stages of design—once a prototype exists—for example in

Authors’ address: Middlesex University, London N11 2NQ, United Kingdom; http://www.cs.mdx.ac.uk.
Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2001 ACM 1073-0516/01/0600–0099 $5.00

improved manuals. Just as good manuals cannot be written for bad systems, many usability-driven design insights come too late to address core issues. At the time of evaluation, prototypes may be considerably advanced in development, so changes suggested by the results of evaluation may be too expensive to implement or strongly resisted because of the investment of time and effort in the existing system.

In this paper, we are concerned with methods to improve usability that can be employed as early as possible in the design process. We propose a tool based on Markov models. The tool is best suited to devices with clear states and transitions: the best, and in fact ubiquitous, applications being push-button devices such as video recorders, mobile phones, ticket machines, and so on [Thimbleby 1992]—and even mouse-driven interfaces, including Web sites. Indeed, for these devices, the creation and analysis of a Markov model can be entirely automated. Thus analysis can be performed as soon as a specification of a user interface has been produced, indeed without the designer requiring any specialist mathematical knowledge to understand Markov models as such. Thus many interface designs can be analyzed quickly and easily to provide quantitative comparisons between designs. Importantly, this analysis can be done before any device functionality has been developed.

Extending the method beyond push-button devices requires care. A Markov model would still produce results, but interpretation of the results may be more difficult. In this paper, to avoid obscuring our technique with secondary issues, we demonsrate it on push-button devices, which is still a worthwhile activity. Push-button devices are widely used throughout the world (e.g., mobile phones), and they are not without their faults.

Understanding the technical details is not required to follow the arguments or examples used in this paper. An important question is whether a method can handle real designs; therefore this paper illustrates the approach by analyzing a variety of real products, such as mobile phones and microwave cookers. We show how variations on the actual designs can be analyzed, and therefore how some new styles of interaction can be explored.

Appendix A provides the code that was used to calculate all examples used in this paper, and it can be used as it stands to replicate our work. A published paper [Thimbleby 1999] explains and provides complete code that can be used for *all of*: Markov model analysis, to run user interfaces, generate user manuals, collect and analyze empirical data from use, and analyze alternate designs. Some details (e.g., the specification of the mobile phone, discussed in Section 3.4) are too lengthy to include here, but full details can be found on the World Wide Web at `http://www.cs.mdx.ac.uk/harold/markov`.

In Section 2, we describe the underlying idea of our approach followed by an informal demonstration of the mathematics involved for a simple device. Section 3 gives further examples on real devices and demonstrates the breadth and flexibility of the technique.

1.1 Contrasts with Other Approaches

Our approach contrasts with cognitive engineering and other approaches to user interface modeling:

—Cognitive models, such as GOMS [Card et al. 1983], can model human interaction with almost any device—since they are concerned with the cognitive issues involved. In contrast, our approach starts with the specifications of artefacts, and we are limited in the sorts of interaction that can be modeled. We cannot handle continuous interaction, but we can handle all discrete systems, whether they are multimodal or involve parallel input and output. Such devices are ubiquitous and pervasive in modern culture.

—Most other approaches only study individual traces of user behavior. Our approach is statistical, and tells us about all possible behaviors of the user. The results of our analyses are expressed in terms like "on average" or can be plotted graphically. Whether the distributions describe individual users over a period of time or a population of users is determined by how the probabilities and models are set up.

—Most other approaches ignore user error or find it difficult to handle. Error is central to a Markov approach. Users can make choices, slips, or errors, with varying probabilities, which Markov models use directly.

—We are concerned with actions a user (or groups of users) takes. Our measures are generally expressed in terms of steps taken to achieve goals (specifically the expected number of steps taken), or of the probabilities of achieving goals. Such measures can be related empirically to times or to aspects of usability.

—Although one can observe users and collect empirical data about their behavior, the complexity of devices usually means that thorough exploration of the possible interactions is infeasible. Devices are complex, have very many features, and users do not have enough time to explore everywhere. The practical consequence of this is that typical user interface design experiments test on lots of users but very few device designs. In our approach, empirical data, such as how likely users are to press buttons, can be used to analyze statistically how users might explore *entire* systems. As usual the better the empirical data, the better the predictions will be—but once a user model has been defined, it can be used using our techniques to explore a device, or a range of devices, more thoroughly than a human user could do (in almost no time at all). Thus it becomes feasible to evaluate a wider range of user interface designs.

2. MARKOV MODELS

Markov models are a standard mathematical technique (see Appendix B for references), and their value for modeling processes has been widely recognized—from describing models for existing systems [Cook and Wolf 1998] to developing test cases [Whittaker and Poore 1993]. Our approach adds a new view of user interface modeling not only internal processes but also of external interaction, which impacts user interface design issues.

In many ways, our approach reflects the arguments that occurred in speech recognition research. For many years, automatic speech recognition systems were "knowledge-based," involving complicated pseudoformal production rules. Not only did these systems have poor recognition performance, errors were difficult to analyze. In the early 1990's, though, hidden Markov models [Young 1993] were built of a variety of speech phenomena, from individual sounds through to intonation patterns [Jones and Woodland 1994]. These models were analyzable (so that improvements could be made systematically), simple and formal and had the great advantage of working very effectively (achieving some 96% recognition on continuous-speech large-vocabulary tasks [Jones and Woodland 1994]).

In addition to providing numerical results, Markov models have a useful property: real systems (e.g., mobile phones) can be modeled effectively in their entirety, with no need to take "abstract," simplifying, or approximate approaches. A Markov model can also be used to build the hardware or to run user interface simulations. They are both powerful enough to be complete, and convenient enough to support detailed analysis. Because of these properties, unlike many "formal methods" for user interface analysis, our approach can be built into design tools. There is no need for the design tool to "understand" how to abstract user interface properties, or to understand how to perform formal specification; conversely, there is no need for the designer to be mathematically sophisticated. Markov models are complete, and any design tool capable of animating a user interface in principle has an exact model available, where our approach can be used directly.

A common criticism of formal methods, and indeed of many other usability engineering methods, is that they depend for their success to an excessive degree on craft knowledge [Stanton and Young 1999]. That is, there are specialized skills tacit in many approaches, and other practitioners therefore find them harder to deploy with good results than their proponents. In contrast, Markov models can be exploited in user interface analysis with no or little craft knowledge. Obviously, craft knowledge can be exploited if it is available, but, for instance, all the results exhibited in this paper can be fully automated. A design tool could, for example, be asked to produce the equivalent of, say, Figure 4 for a working design, and it could do so with no further direction.

Any model is not worth its salt unless it provides useful measures at a timely point in the design process. As with all formal representations, however simple, Markov models provide the facilities for measuring the model. And like all models, translating the results back to the thing modeled requires interpretation. The further the model is removed from the original, the more care must be taken in the interpretation. To avoid becoming entangled in difficult and ambiguous interpretations that are secondary to the methods being demonstrated, the examples in this paper steer away from tasks and goals and instead make direct mappings between system states and model states.

2.1 Taming Finite-State Machines

Behind any usability analysis there must be some model both of the device being modeled and of the user. Finite-state machines (FSMs) are particularly effective at modeling certain devices as they are simple, quick to produce, and scalable to any size. Also, as well-defined mathematical objects, it is possible to perform complex reasoning on the model to produce reproducible, quantitative results. We add a Markov model to the FSM that assigns a probability to transitions between states and analyze the Markov model to provide insights into the user interface design.

Like any model, finite-state machines are necessarily incomplete. They provide only a simplified representation of user interaction; there is a body of opinion that FSMs are inappropriate for realistic interactive systems [Kieras and Polson 1985; Monk and Curry 1994; Newman and Lamming 1995; Palanque and Paternò 1998; Wasserman 1985]. FSMs are described in elementary theory of computation books (e.g., Lewis and Papadimitriou [1998]), and often covered briefly before moving on to the more powerful but still impractical Turing Machine. Indeed it is a standard result that FSMs are not very powerful computational devices. But to understand these sorts of results as implying that FSMs are inadequate would be mistaken; Feynman [1996] is one of the few authors to make this elementary point clear. Their very simplification makes FSMs a flexible tool suitable for highlighting fundamental, structural errors in a user interface.

Finite-state machines have the problem that the number of states for even modest systems is enormous. We use transition diagrams to illustrate some of our examples, but transition diagrams or indeed any representation (including Statecharts [Harel 1988; Harel and Politi 1998]) rapidly become too big to be useful. The Markov model however is fully scalable, and so functions just as well on large FSMs as small ones; using it does not require drawing a diagram. It tames FSMs of all sizes and produces meaningful statistics.

The Markov models themselves do not get out of hand, as their underlying representation is a matrix representing the probability of transitions between states. The number of nonzero entries in each row of the matrix is determined by the number of possible transitions. With push-button devices, these transitions are initiated through button presses, so there are generally very few possible transitions in comparison with the possible number of states. The resulting matrix is sparse, and there are many well-known methods for storing, representing, and manipulating sparse matrices, as well as techniques specifically for handling large Markov analyses—systems of billions of states can be handled (see Section 2.3).

Halbwachs [1993] showed that FSMs are sensitive to small changes in the high-level definition. However, as we automate the production of the FSM from the high-level description, there is no extra burden on the designer, and the Markov model can be automatically adapted accordingly.

It is beyond the scope and purpose of this paper to defend FSMs further, except to mention that they are adequate to handle multiuser and concurrent systems. For example, there are many high-level concurrent programming languages that compile into FSMs (such as Halbwachs [1993] and Magee and Kramer [1999])—thus neither designers nor developers need be concerned with the details. Although it can be done directly, the simplest way to adopt the Markov approach to concurrent systems is to specify systems in appropriate high-level languages that compile into representations (such as FSMs) that can then be analyzed directly.

Table I. Preliminary Definition of Torch

State	Press `ON`	Press `OFF`	Remove battery	Add battery
1: Switched on, with battery	1	2	3	1
2: Switched off, with battery	1	2	3	2
3: No battery	3	3	3	?

2.2 From Finite-State Machines to Markov Models

We now explain how Markov models work. We start with a discussion of simple FSMs and generalize them to probabalistic models.

Though finite-state machines are a simplification of user interaction, their simplicity makes them a flexible tool. Further examples of their use can be found elsewhere [Thimbleby 1994; 1997].

A push-button device can be represented as a finite-state machine, where each state is something distinctive the device is "doing" (possibly including being off). Pressing buttons (or perhaps doing other things) on the device changes its state or possibly leaves it in the same state. For any particular state, each button always does the same thing. If somehow a button does different things in the same state, then we are mistaken in assuming it is the "same" state. However, we can choose to label by a single state different things the device does: for example, we might decide, that for some purposes, what time a clock shows does not matter—all times could then be classed as one state. For many tasks, whether a clock shows 17:28 or 17:29, say, is immaterial: for an analysis we may only be interested in whether the clock is running or not, or perhaps whether it is displaying a morning or afternoon time.

The states are numbered $1, 2, 3, \ldots N$. In principle we can do this for any sort of device, and when suitable design tools are used no manual work is involved in counting off the states.

As a concrete but simple example, consider a simple battery torch, with two buttons `ON` and `OFF`. In principle there are numerous states: the torch may have no batteries, or it may have dead batteries; it may have one missing; it may have a dud light bulb; it may be on, and so on. Clearly, some of the states we can imagine will not be sufficiently interesting to be considered distinct. Table I shows a preliminary definition of a simple torch, where the number of states has been tentatively fixed at 3.

There are some interesting problems visible already. For example, how can you add a battery to a torch that already has a battery? Should we add more states to cater for "squashed battery," "broken torch," or "frustrated user"? Once the table starts to get realistic and represents more and more of the world, it may be hard to know where to stop. For the time being, we will only be interested in successful operations (i.e., only inserting a battery when that is possible). There is another problem, indicated by the question mark in the table. If a battery is inserted when in state 3, the bulb will either come on or stay off. Which? This is the situation alluded to above: state 3 is in fact two states: no battery and off; no battery and on. For many design purposes, having

Table II. Refined Definition of Torch

State	Press ON	Press OFF	Remove battery	Add battery
1: On, with battery	1	2	3	1
2: Off, with battery	1	2	4	2
3: No battery (on)	3	4	3	1
4: No battery (off)	3	4	4	2

a consistent model is more important than having a complete one—not that it is feasible to have a complete model of a human-machine system. Table II shows a refined version.

Such tables describe a device in terms of states and actions, such as button presses, depending on the device details. (Another example is provided in Figure 1.) If there are N states and B actions, the tables have $N \times B$ entries. The *adjacency matrix* (also known as the state transition matrix) is an alternative representation that ignores the details of which actions are used. If there are N states, an adjacency matrix A is an $N \times N$ matrix where

$$A_{ij} = \begin{cases} 1 & \text{if there is a user action that changes state } i \text{ to state } j \\ 0 & \text{otherwise} \end{cases}$$

Table II can be represented as a 4×4 adjacency matrix:

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

Coincidently this matrix is the same size as the table,[1] but it is to be read differently. Each row is a state-from, and each column is a state-to. Thus row 1 indicates there are possible transitions from state 1 to states 1, 2, and 3, but not to state 4.

Notice the abstraction of the user interface as represented by the matrix: the matrix certainly does not tell us everything about the design. Nevertheless, the matrix has an interesting structure, even though one cannot tell from the matrix what any of the operations are; in fact, the matrix does not show what the names of the states are either.

2.3 Concrete Issues of Scalability

A simple digital alarm clock has states for 24 hours and 60 minutes for the time of day and for the time of the alarm, and all times two for the possibility that the alarm is either enabled or disabled: that is $24 \times 60 \times 24 \times 60 \times 2 = 4{,}147{,}200$ states in all. If this FSM was represented as a simple adjacency matrix (as we can do explicitly for the torch) it would have over four million rows and columns, and without compression it would require about 2,000 gigabytes of computer memory! Fortunately, the actual size of the FSM is irrelevant to the

[1]In this example, there happens to be the same number of actions as states, 4 each.

mathematics. According to Shneiderman [1998], it would be "a major contribution" to have scalable formal methods and automatic checking of user interface features: finite-state machines (and Markov models) have this advantage.

For a device with b buttons, no row in the matrix has more than b nonzero entries. For example, if we replace the two torch buttons (ON and OFF) with a single push-on/push-off button, its matrix would have one fewer nonzero entries per row:

$$A' = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

From this new matrix, incidentally, it is clear that *any* action the user does always changes the state: the leading diagonal of the matrix is all zeroes. Pressing the push-on/push-off button always changes the state, and (as before) successfully removing or adding a battery also changes the state.

Indeed the number of buttons is usually kept relatively low. Even a general-purpose computer only needs about 50 keys (a QWERTY keyboard) to operate. Thus the matrix is sparse (mostly zero), and as the number of states increases the matrix becomes exceedingly sparse. Furthermore, buttons usually do something reasonably consistent (e.g., the off button always makes the same state transition). Typically, then, a large adjacency matrix can be easily treated as a set of rules rather than as an explicit matrix with every element taking up memory, so it can be quite compact—in fact, if there is *no* way to represent the matrix compactly, there is probably no way to make the device easy to understand for a user, and certainly no way to program it systematically. See Baldi et al. [1999] and Deavours and Sanders [1998] for discussion of symbolic and "matrix-free" methods, which provide the analytic power without the cost of large, explicit matrices.

2.4 Introducing Empirical Data

We have seen how an adjacency matrix can represent (an abstraction of) a finite-state machine, but it "knows" nothing about the user. But we might know, for instance, that users are more likely to add batteries to a torch when there are no batteries in it than when the bulb is on. The adjacency matrix tells us that a user can remove batteries from a torch that is on—it does not say whether this is likely to happen. As a next stage in realism, we introduce the probabilities of the user making the device change state. We refine the adjacency matrix A to the *transition probability matrix*, P.

Continuing with the torch example, below we give some example values for P (which in this case, we have made up).

$$P = \begin{pmatrix} 0 & 0.9 & 0.1 & 0 \\ 0.8 & 0 & 0 & 0.2 \\ 1 & 0 & 0 & 0 \\ 0 & 0.1 & 0.9 & 0 \end{pmatrix}$$

When A_{ij} is zero, P_{ij} is zero, because knowing something about the user's probable behavior cannot make the device do things it cannot do. But the probabilities add more information.

The user may not do some things for very long times, if ever, but so far as the outcomes represented in the matrix are concerned there are no other possibilities than to enter one of the four states. Necessarily, each row in the transition matrix sums to one: the user does *something* with probability one.

The probabilities here show that a user is more likely to switch a torch off than to remove its batteries. If the torch is on, with probability 0.9 the next change of state will be the torch being off, and with probability 0.1 the user might remove the battery. Or if the torch is off with no battery in it, with probability 0.9 the user will try to switch it on (though, of course, the torch will not light). We could make the probabilities more realistic, for instance by collecting data on actual users' behavior—however, we are using the example to explain the method, not to analyze torches or their use in depth.

At this stage it may appear that there is no obvious advantage in using a matrix to represent the probabilities: it looks like maths for its own sake. This is not the case; in fact matrix algebra is very useful.

Consider that a user may pick up a new torch in state 4 (no battery and off). What will happen next? By inspection of the matrix, we can see the torch will go to state 2 with probability 0.1 and state 3 with probability 0.9. In general, at any time the torch has various probabilities for being in each possible state; we can represent these probabilities in a *state probability vector*, v. When a new torch is picked up, off with no batteries in it, the probabilities are $v_0 = (0\ \ 0\ \ 0\ \ 1)$, and after an operation they are $v_1 = (0\ \ 0.1\ \ 0.9\ \ 0)$—these figures are determined by the probabilities in the matrix we have already established. In matrix algebra we have simply $v_1 = v_0 \times P$, i.e.,

$$(0\ \ 0.1\ \ 0.9\ \ 0) = (0\ \ 0\ \ 0\ \ 1) \times \begin{pmatrix} 0 & 0.9 & 0.1 & 0 \\ 0.8 & 0 & 0 & 0.2 \\ 1 & 0 & 0 & 0 \\ 0 & 0.1 & 0.9 & 0 \end{pmatrix}.$$

In general, we can find the probabilities for any time in the future:

$$\begin{aligned} v_2 &= v_1 \times P = v_0 \times P \times P \\ v_3 &= v_2 \times P = v_0 \times P \times P \times P \end{aligned}$$

and so on. In general, since repeated multiplication is the same as raising to a power we have

$$v_n = v_0 \times P^n.$$

The value of this mathematical representation is easy to demonstrate. Consider that the user has choices at each operation, and as time progresses there will be more and more ways the user could have used to have reached any goal state. Indeed after just 10 operations for this simple device, as specified above, there will have been 1,024 different ways to return to state 1, or after 5,000 operations there will have been a staggering 10^{1505} different ways. Working through the probabilities of all these cases is impractical if done explicitly, but if we want to know, say, what the probability of being in the first state is after 5,000 steps is, we just calculate P^{5000}, and moreover this calculation may be done efficiently: there are standard ways of raising to a power so that P^{5000} only requires 16 matrix multiplications (much less than the 4,999 that might have been expected [Knuth 1998]). Furthermore this simple calculation represents *all* of the 10^{1505} different ways the user may have got to this state! Thus we see the advantage of the mathematical approach—a concise description and an efficient means of working things out.

If we assume that P does not change, then this is a so-called homogeneous Markov chain. Many mathematical results are known for Markov chains.

2.5 Usability

To assess the usability of a device, among other factors, we want to know how hard (in some sense) the device is to use. The number of state transitions a user takes to achieve some task is an obvious and simple measure of difficulty. If the number is huge, possibly the user will not live long enough, and the device will be impossible to use; or if the number is zero, the device is telepathic and (unbelievably) easy to use! For intermediate numbers, the more transitions required make the device harder to use. Transitions may take different amounts of time; for a device like the torch, inserting a battery takes longer than pushing a button.

There have been many studies of time to perform keystroke tasks (e.g., Silfverberg et al. [2000], which studies timings for mobile phones), usually under the assumptions of error-free performance. However, because we are using Markov models, we are not restricted to considering only "error-free" behavior on the user's part. In particular, the sentiment that a user taking a long time has more time to make errors and so takes even longer is already built into the approach; in fact, as our examples below show, the damaging effects of errors on user performance are starkly revealed. Section 2.7 discusses one way of introducing knowledge of correct operation, and we shall see that our approach gives a very good way of visualizing the impact of knowledge on usability.

Given the transition probability matrix P of some device, we want to be able to answer questions like "If a user starts in state i, how hard is it to get to state j?"

Because P is a transition probability matrix, the "how hard?" question translates more precisely to "what is the mean number of transitions to first reach state j starting from i?" This is easily answered by a Markov chain model. Appendix B derives a formula for obtaining this number from any matrix P. The next section discusses what we can do with the number. (Many other measures can be obtained from a Markov model, but one example will do for the purposes of this paper.)

In practice P can have probabilities chosen to suit the known characteristics of the user interface, or of a user's behavior (for certain tasks, or averaged over a suite of tasks). We can most easily assume all button presses are equiprobable (as we do in all subsequent examples below)—this represents the “walk up and use” case well for some devices. Our analysis is here restricted to homogeneous Markov processes, meaning that the probabilities are not conditional (e.g., on the user’s prior actions or what they learn); but the probabilities can be different for different states or buttons—say, if some are physically larger, or lit up in certain states. Setting the probabilities to reflect this is straightforward (e.g., see Section 3.4).

In real-user-based usability simulations, designers are able to trace button press events—and hence calculate average goal completion efforts and times, so that once a prototype device has been built and can be used, transition probabilities can be acquired from actual user behavior. A Markov model provides a good way of recording the behavior. Section 3.1 shows that simulation and analysis are easily combined: the analytic approach can easily be simulated; the simulation data can easily be combined with further analysis, and so on. Moreover a random approach (e.g., simulating a random user) has been shown to be very good at identifying programming problems [Miller et al. 1990], and therefore would be of use when the user interface was not automatically derived from the specification.

2.6 Knowledge as a Function of Cost

The so-called “cost-of-knowledge characteristic function” has been used to represent the cost of acquiring knowledge about a device [Card et al. 1994]. If we represent cost by the number of buttons pressed (which is straightforwardly related to time), and knowledge by the number of states visited by the user, then the cost-of-knowledge graph plots the maximum number of states that can be visited by at most a given cost. The graph can be plotted from empirical data (as is done by Card et al. [1994]), by assuming the user knows exactly how to use the device, or—more realistically, allowing for errors—taking the cost from a Markov model [Thimbleby 2000].

Although such graphs are useful in design (e.g., to guide reduction of average costs of tasks) they might better be called cost-of-access graphs, since they measure state accessibility. It is unlikely that a user’s knowledge about a device increases linearly with merely accessing states. Moreover, knowledge of states (what a device does) is not knowledge of use (how to get it to do those things).

2.7 Usability as a Function of Knowledge

To get usability metrics we need to consider the user’s knowledge of how to use a device. Our approach is to take a mixture of a “perfect” error-free approach of performing a task with the original “ignorant” P. We define the user’s “knowledge factor” k as a number $[0\,..\,1]$, essentially a probability they will behave like a fully knowledgeable user (e.g., a designer!) rather than randomly. If $k = 1$, the user’s knowledge is equal to a designer’s, and they operate the device optimally; if $k = 0$ the user has no knowledge, and they act randomly with no preferences, and are equally likely to press any button.

The perfect approach is what a designer knows is the best way of performing a task. This is easy to determine from P, by taking the shortest allowable sequence of transitions, and setting those transitions with a probability of one. (Shortest-path algorithms are well known; *Mathematica* provides one that can be used directly on a transition matrix.)

The transition probability matrix for a user with knowledge k is then $kD + (1-k)P$, where D is the "perfect knowledge" (designer's) transition probability matrix (with 1's on the optimal transitions) and P the original "knowledge-free" matrix. As we have defined it here, the knowledge k measures the knowledge to achieve a certain task optimally; modeling general knowledge of a device would require a set of matrices, D_g, one for each goal.

We can now easily work out the usability of a device in terms of the user's presumed knowledge. We can plot graphs of expected task completion cost against k. We can also view k as representing the user's accuracy: if we assume the user does know exactly what to do, then the lower k, the more errors they are making—until at $k = 0$ they are making so many errors that they are behaving randomly.

Although our approach in this paper uses Markov models, this is not essential for viewing usability as a function of knowledge. We could use cognitive models, or any other models that provide numbers. However, an advantage is that we can easily take combinations of models (in this case, linear combinations of D and P) without any artificial manipulations.

3. WORKED EXAMPLES

If we reanalyzed a system that we knew had a usability problem, methodologically we could be criticized that we knew what sort of problems we were looking for. That would leave open whether such an approach was useful in the design process *before* problems are recognized. Instead, for our first example we take a specification directly from Sharp [1998] in interaction design. Sharp's thesis was concerned with the reliability of on-screen simulations for usability studies (he was more concerned with photo-realistic 3D models, rather than outline 2D models such as our simulation), and was not concerned with the usability of devices *per se*.

After analyzing variations on the Sharp design (Section 3.3), subsequent examples are a mobile phone (Section 3.4), the Panasonic *Genius* digital clock (Section 3.5), and a combination lock (a device intended to be hard to use) (Section 3.6).

3.1 A Microwave Cooker and Its Implementation in *Mathematica*

Sharp's rules for his microwave cooker are shown in Figure 1. From this, some simple manipulation in *Mathematica* gives us a finite-state machine representation sufficient to perform analysis (complete details are given in Appendix A).

Mathematica (Version 3 and later) [Wolfram 1996] has a simple and convenient scheme whereby buttons can be bound to arbitrary actions. This is a standard technique to implement "easy-to-use" *Mathematica* packages: we simply exploit it to provide the user interface for the required device (Appendix A.2

	States					
Buttons	`Clock`	`QD`	`Timer1`	`Timer2`	`Power1`	`Power2`
Clock	Clock	Clock	Clock	Clock	Clock	Clock
QD	QD	QD	QD	QD	QD	QD
Time	Timer1	Timer1	Timer2	Timer1	Timer2	Timer1
Clear	Clock	Clock	Clock	Clock	Clock	Clock
Power	Clear	QD	Power1	Power2	Power1	Power2

QD = Quick Defrost

Fig. 1. Specification of Jonathan Sharp's microwave cooker. Columns are states the device is in; rows are buttons. By knowing the current state and a button, the table gives the next state. Thus, pressing the `Quick Defrost` button (i.e., row 2 of the matrix) when in state `Clock` causes the device to enter state `Quick Defrost`. As in Sharp's original specification, notice that `Quick Defrost` is both a state name and a button name.

provides the complete code we used to create the user interface, and which will work with any device). The *Mathematica* simulation of this microwave cooker is shown in Figure 2. With the simulation, we can also perform conventional usability studies and obtain empirical data that could refine the probabilities used in the mathematical analysis.

The adjacency matrix for Sharp's microwave cooker is

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

We can now obtain (one line later in *Mathematica*) the all-pairs shortest paths (the quickest ways to get from any state to any other state):

$$\begin{pmatrix} 0 & 1 & 1 & 2 & 2 & 3 \\ 1 & 0 & 1 & 2 & 2 & 3 \\ 1 & 1 & 0 & 1 & 1 & 2 \\ 1 & 1 & 1 & 0 & 2 & 1 \\ 1 & 1 & 2 & 1 & 0 & 2 \\ 1 & 1 & 1 & 2 & 2 & 0 \end{pmatrix}$$

For example, to get from either state 1 (`Clock`) or state 2 (`Quick Defrost`) to state 6 (`Power2`) takes three button presses. The leading diagonal is all zeros because it takes no steps to get from a state to itself.

The entries in this matrix are the button-pressing costs for an error-free user who knows what they are doing. Such an error-free user must be able to identify each state correctly (or at least be able to correctly identify the starting state for any task, and then know what to do from memory).

Evidently, for this device a user can get from any state to any state in at most three button presses: the device looks easy to use. The shortest paths are routes that are obvious to designers, especially if they use tools to visualize the state diagram (see Figure 3, which, though tidied up for typesetting this

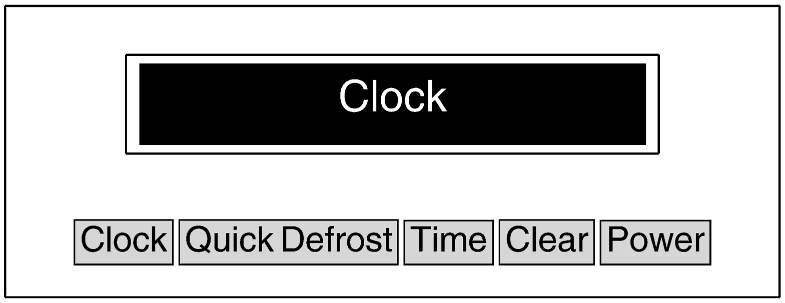

Fig. 2. Screen shot of a *Mathematica* user interface simulation, showing screen display and button arrangement. The device is currently in the clock state. The buttons can be derived directly from the button specification. Clicking on a button calls the *Mathematica* function `press[`*`name`*`]`, which changes the device state and causes the display to change appropriately. Code for this simulation is provided in Appendix A.

paper, is in the form drawn by *Mathematica*). Note that it would be easy to demonstrate such a device and give a persuasive impression that it was easy to use [Thimbleby 1996]. However, the shortest-path numbers are much smaller than a Markov model predicts—e.g., for this device getting from, say, `Power1` to `Power2` takes 120 steps. This is so large it deserves explanation:

—A Markov model does not "realize" when it is going around in circles, so it gets stuck in repetitive behavior easily. (In other words, a homogeneous Markov model does not learn.)

—The button-pressing probabilities used in the example are all equal, regardless of state. A more accurate model would use more realistic probabilities (e.g., as measured from real users' behavior).

—Designers tend to seriously overrate the ease of use of their system, even for something as clear-cut as Sharp's microwave cooker.

Humans can also "loop" when they do not fully understand a device. An all-too-familiar anecdote will suffice to illustrate the problem. Recently, one of the authors and his 17-year-old son were trying to program their Goodmans VN6000 video recorder following a power cut, which had reset the date and time.

The task, therefore, was to reset the clock to the current time, then use the VideoPlus+™ system for entering the code for the program they wanted to record.[2] Setting the clock was easy: the remote control has a menu button, and all subsequent interaction was through a television on-screen dialogue. Then the VideoPlus+ code had to be entered. However the on-screen menu

[2]VideoPlus+ uses a number (typically printed along with the television program listings) to specify the channel, date, and start and end times of a broadcast program. Given how hard most video recorders are to program, entering a single, humanly meaningless number of many digits is easier than working out how to enter the different parts of the timing information separately. The VideoPlus+ code is a hash code of the information, and is designed so that more common timings have shorter codes. For example, 258 is the code for channel 1, 18:00–18:30 on 19 April, but 46140247 is the code for channel 5 at 00:45–03:45.

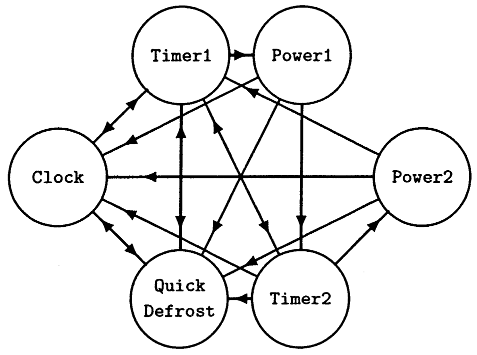

Fig. 3. Sharp's microwave cooker, drawn as a directed graph. This is a ranked embedding: states drawn in the same column take the same number of button presses from (in this case) the `Clock` state. (So `Power1` takes 2 steps to reach from `Clock`, as does `Timer2`, and `Power2` takes 3 steps.)

only had "`VIDEOPlus+ PRESET`" as a choice, which was explored. This seemed to allocate VideoPlus+ codes to television channels, and was apparently correctly set despite the power cut. There was a menu choice for "`CHANNEL PRESET`," and this set UHF channels to television channels, and this too was correctly set. Perhaps the VideoPlus+ system only worked when it "knew" the clock was running? Having successfully set the clock, the users therefore switched the recorder on and off, and tried again.

Despite the "user team" including a teenager (teenagers are supposed to understand these things), both users "looped" for about 40 minutes before despairing and hunting down the video recorder manual.

The correct (that is, the manufacturer's idea of "correct") operation was to press another button on the remote control labeled "`VIDEO Plus+`"! The users had not spotted it because they had become drawn into the many on-screen choices offered by the menu system.

A Markov model of this interaction would also have taken a long time to solve the problem, because it would have spent a similarly proportionate time in the menu system. A designer should therefore consider the large numbers Markov models typically generate as significant clues to improving designs.

3.2 Knowledge/Usability Graph for the Microwave Cooker

We take the task to get from `Clock` to `Power2` state in Sharp's microwave cooker and examine the knowledge/usability graph. Recall that the graph shows the expected time given a transition probability matrix $kD + (1-k)P$, where D is a "perfect knowledge" (designer's) matrix (with 1's on the optimal transitions) and P the original "knowledge-free" matrix. The values of P (random use) and D (perfect, or designer's, use) used are shown below, and the resulting graph is shown in Figure 4 (solid line).

The probabilities in P are calculated assuming button presses are equiprobable: for this device there are 5 buttons, each pressed with probability $^1\!/_5$. Some buttons will leave the state unchanged, so the leading diagonal of the matrix has some elements greater than $^1\!/_5$.

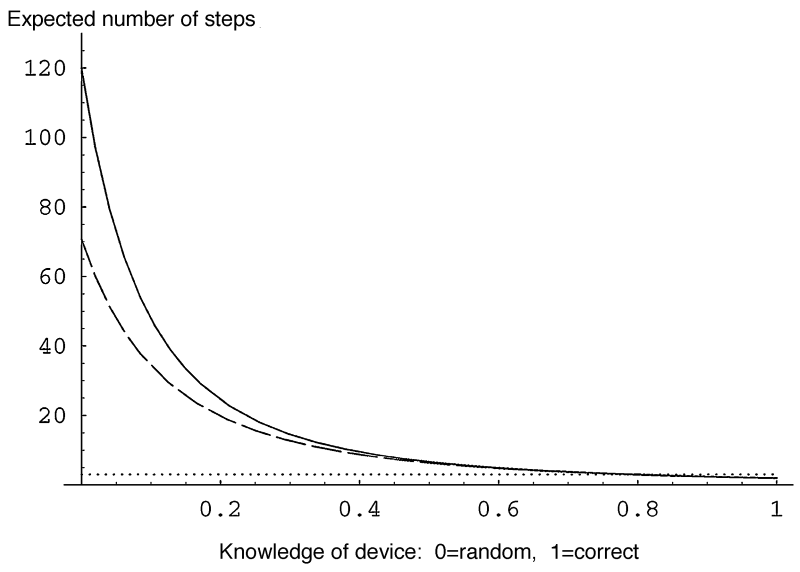

Fig. 4. Plot of expected number of steps against knowledge of device. The task is to get from `Clock` to `Power2` state. The solid line is the original microwave cooker, and the dashed line is the LED-enhanced device. The horizontal dotted reference line (expectation = 3) is the optimal task completion cost, i.e., assuming the user knows exactly how to do it and makes no mistakes.

$$D = \begin{pmatrix} 0&0&1&0&0&0\\ 0&0&1&0&0&0\\ 0&0&0&1&0&0\\ 0&0&0&0&0&1\\ 0&0&0&1&0&0\\ 0&0&0&0&0&1 \end{pmatrix} \quad P = \begin{pmatrix} 3/5&1/5&1/5&0&0&0\\ 2/5&2/5&1/5&0&0&0\\ 2/5&1/5&0&1/5&1/5&0\\ 2/5&1/5&1/5&0&0&1/5\\ 2/5&1/5&0&1/5&1/5&0\\ 2/5&1/5&1/5&0&0&1/5 \end{pmatrix}$$

It is important to recall, that (by simple programming in *Mathematica*) these matrices are generated automatically, and that changing the definition of the device changes the matrices automatically. Again, the approach scales up to more complex devices than we can conveniently show in this paper (later sections discuss a larger device).

The downward slope of the graph (Figure 4) is not surprising, but the shape of the curve is interesting. It can be seen that Sharp's microwave cooker is good in the sense that a little gain in knowledge initially gives a rapid improvement in task completion performance. Also, good but imperfect knowledge has comparable performance to perfect knowledge. In other words, most users get satisfactory performance with only a casual knowledge of perfect use.

3.3 Analyzing Variations of Sharp's Microwave Cooker

The "random" user represented by P knows nothing about the device. How much would it help if, say, there were LEDs on each button, at least telling a user that these are the buttons that do *something*? So, whatever the user wants to do, this device ensures they do not need to waste time pressing useless buttons.

Some devices, such as video recorders, which are often used in the dark, would certainly benefit in other ways as well—the user could easily locate functional buttons. Alternatively some mechanical arrangement might actually hide the nonfunctional buttons. A new matrix P_{LED} can be defined to reflect random use of this modified design:

$$P_{\text{LED}} = \begin{pmatrix} 0 & 1/2 & 1/2 & 0 & 0 & 0 \\ 2/3 & 0 & 1/3 & 0 & 0 & 0 \\ 2/5 & 1/5 & 0 & 1/5 & 1/5 & 0 \\ 2/5 & 1/5 & 1/5 & 0 & 0 & 1/5 \\ 1/2 & 1/4 & 0 & 1/4 & 0 & 0 \\ 1/2 & 1/4 & 1/4 & 0 & 0 & 0 \end{pmatrix}$$

Although Sharp's microwave cooker has some explicit self-transitions, the diagonal of P_{LED} is entirely zero: the LEDs only light when a button changes the device's state.

A plot of this shows that the modified device (Figure 4, dashed line) is an improvement over the original. This is not bad for a device that does not know what its user is trying to do! Interestingly, it does not give such a fast rate of improvement as the user learns how the device works. For more complex devices we would expect even better improvements if LEDs were used—and, we would certainly expect the little LEDs to help sell the device in a shop! However, this simple analysis demonstrates the ease with which we can analyze and compare variations on a design.

Now consider modifying the device so that, rather than lighting LEDs on buttons that work, buttons *always work*. We can arrange that buttons not defined by Sharp return the device to `clock`. A designer might decide that this is a good idea because any "incorrect" operation of the device would return it to a safe, well-defined state.

$$P_{\text{Always}} = \begin{pmatrix} 3/5 & 1/5 & 1/5 & 0 & 0 & 0 \\ 4/5 & 0 & 1/5 & 0 & 0 & 0 \\ 2/5 & 1/5 & 0 & 1/5 & 1/5 & 0 \\ 2/5 & 1/5 & 1/5 & 0 & 0 & 1/5 \\ 3/5 & 1/5 & 0 & 1/5 & 0 & 0 \\ 3/5 & 1/5 & 1/5 & 0 & 0 & 0 \end{pmatrix}$$

Modified in this way, Sharp's microwave cooker takes 129.167 steps to get from `Clock` to `Power2`, when $k = 0$. This is only a bit worse (3%) than the original version, but 67% worse than the LED version. (As before, since it is the same underlying FSM, it must take 3 when $k = 1$.) This alternative design should not be preferred over the LED idea, but whether it is an improvement over the original probably should not be answered just by considering Markov models—there may be psychological reasons to prefer or reject it.

Of course, with other devices, the comparisons would come out differently: we are not concluding that LEDs *always* improve devices significantly, or that the "no incorrect operation" style always makes a small improvement. What we have shown, though, is that making such comparisons is very easy.

The JVC UX-T20B stereo system is a simple commercial example of using LEDs in this way. Its "Compu Play" feature means that when in standby only one button press is needed to start the system playing either tuner, compact disc, tape (or auxilliary input) depending on which button is pressed. (The power switch, then, is only needed when the user is not going to play something immediately but instead, for example, set the alarm.) Its "Illumi Magic" feature is linked to an infrared sensor, so when an object approaches the sensor, the buttons that activate the Compu Play feature are highlighted with an LED strip near the appropriate buttons. Thus, by vaguely waving a hand at it, it is enough to light the LEDs and, from that, very easy to start the machine playing in whatever mode is required, even when the machine is close to the floor in a dark corner of a room.

Markov models expose the problem of "looping"—of a user going around in circles and not making progress toward their goal. The LED-based device was easier to use because a large number of short loops were eliminated from the user interface. At the other extreme, the entire device could be a single loop, of maximal length. It would then be a *ring*. Rather than use push buttons, a ring has better affordance implemented using a rotary knob.

Suppose a knob is used as the user interface to Sharp's microwave, and that (due to its design features) it is turned clockwise with probability p and counterclockwise with probability $1 - p$. The transition matrix is

$$P_{\text{Ring}} = \begin{pmatrix} 0 & p & 0 & 0 & 0 & 1-p \\ 1-p & 0 & p & 0 & 0 & 0 \\ 0 & 1-p & 0 & p & 0 & 0 \\ 0 & 0 & 1-p & 0 & p & 0 \\ 0 & 0 & 0 & 1-p & 0 & p \\ p & 0 & 0 & 0 & 1-p & 0 \end{pmatrix}.$$

The rather attractive Figure 5 shows the expected least cost of reaching, from any state, each of the device's five others.[3] Interestingly, the worst average cost (over all *six* possible goals) arises when $p = 0.5$, which corresponds to an unbiased knob. From a purely Markovian analysis, a one-way knob would be more efficient to use ($p = 0$ or $p = 1$). Clearly, though, a user's actual performance would be more sophisticated, and a one-way knob would be suboptimal. Nevertheless, the costs are so much lower than for a push button user interface that there could be merit in considering this style of interface further—particularly for "walk up and use" interfaces.

A ring has only one loop, but some states are far away. Another radical alternative is a *star*. A star has one state in the middle, and the others radiating off it. Each of the 5 noncentral states has a single transition, back to the center. Thus, this design requires six buttons: five to get from the center to each of the other states, and one to return from them to the center state. Designed like this, the average cost to reach any state from the center is 9, and from

[3]Figure 5 in fact shows all 6×6 graphs of getting from any knob position to any other. Six of the graphs show the expected least cost of getting from a state to itself, which is zero. Of the other 30, there are five groups of six, each the same (because of symmetry). Hence the figure shows five curves.

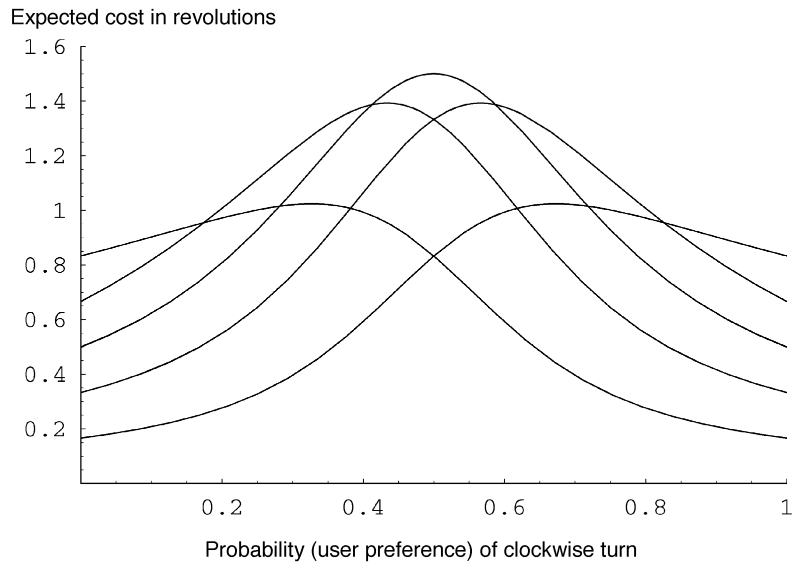

Fig. 5. Expected cost of using a knob to achieve the five states of Sharp's microwave cooker, starting from any given state, plotted against probability of rotating the knob in a specific direction. The cost (vertical axis) is measured in total revolutions assuming the six knob positions are equally distributed around a circle—multiplying the scale by 6 makes it the number of "clicks."

any noncentral state, 10. This is a bit worse than the ring, but better than the original design—at least, so far as the numbers are concerned!

A star suffers from having labeled buttons that only work some of the time. The "return to center" button only does anything if the device is not already at the center state; and the other five buttons only work when the device is in the center state. Worse, these five buttons do not always *leave* the device in the appropriate state when they are pressed.

If six buttons are going to be used, as there are six states, then the complete symmetric digraph (K_6^*) is a natural design choice: there is a transition from every state to every other state. The transition probability matrix is trivial: every element is equal to $^1\!/_6$. The average cost to reach any state is now 6, or to be precise—if a user walks up to the device, not knowing what state it is in, or if they do not bother to check and *always* press at least one button—the average cost is 5.2 presses—and that is if $k = 0$. When $k = 1$ (i.e., the user knows what they are doing), the K_6^* design takes 0.8 presses on average (1 press in the worst case), which is much faster than Sharp's original, which takes 1.2 presses on average (3 presses in the worst case). That seems like a useful gain at the cost of only one button; and it is not just a numerical gain, since the buttons *always* leave the device in the appropriate state—this design is both faster, and it is mode-free.

3.4 A Larger Device: The Nokia 2110 Mobile Phone

Sharp's microwave cooker has only six states, and the analysis is not intrinsically difficult. We now examine the function menu of a Nokia 2110 mobile phone. This part of the phone has 152 states.

The function menu of the Nokia 2110 is controlled by four buttons, two "scroll" buttons that move up and down in a menu, and two selection buttons that have changeable ("soft") meanings as shown in the phone's display panel. Initially, these two buttons show `Menu` and `Memory`. The user would press `Menu` (which then disappears), and the other button becomes `Quit`. When the user scrolls up or down the menu, the first button becomes `Select`, which either selects the function shown in the menu, or selects a further submenu. The `Quit` button quits each level of the submenus, taking the user back to the position where that menu was selected. The scroll buttons can be pressed repeatedly and cycle through any given level of the menu hierarchy. There are a few other features, such as being able to press numeric keys to select functions faster than searching for them step-by-step in the hierarchy.

For reasons of space we do not show the data here. Using a basic probability matrix of equiprobable button presses, we find the cost to be 602,235 (for a random user) for the task of setting selecting "incoming calls" (a selection in the "call barring" menu, itself a selection in the "security options" menu), starting from standby. This number is so large because of the effect of the `Quit` button, which the knowledge-free model repeatedly uses, and therefore hinders reaching any desired goal.

The knowledge/usability graph as used in the analysis of the Sharp microwave cooker showed how easy to use the device was against how much the user knew how to use the device for the task in question. For the Nokia mobile phone, what would happen if we modeled a user whose knowledge was to avoid using the `Quit` button? In other words, the knowledge k is the "knowledge" to use the `Quit` button. Figure 6 shows the resulting graph for the Nokia 2110.

At $k = 0$ (to the left of the graph), the user never uses `Quit`—and therefore cannot correct any errors. With $k > 0.2$ the difficulty of using the phone increases dramatically. Here we are seeing that if the user presses `Quit` too often they cannot make effective progress.

There is an optimal use of the `Quit` button, around $k = 0.037$. As a hint to the designer of the mobile phone, this could be taken to mean make the `Quit` button much smaller, so the user tends to use it less than other buttons. To do this on a device like the Nokia 2110, where the `Quit` key is a soft multipurpose key, could cause problems in other parts of the user interface. However the Nokia 2110 has a much smaller button `c`, normally used for correction (and for returning to standby when in the menu hierarchy). This button could have been used with the same meaning as, but replacing, the current `Quit` key.

If the `Quit` key is so critical, which it certainly is compared to the other menu-selection keys, then the user interface design that makes it so should be examined closely. It may be that the method of quitting submenus (i.e., the structure of the user interface) is the problem, not the frequency of use of the `Quit` key itself.

In many ways the problem with this sort of analysis—and certainly for picking examples to illustrate its use in a paper such as this—is the whole range of new questions that are immediately suggested!

Given that *Mathematica* had been set up for the analysis, the only hard work in analyzing the Nokia phone was working out the transition matrix. This was

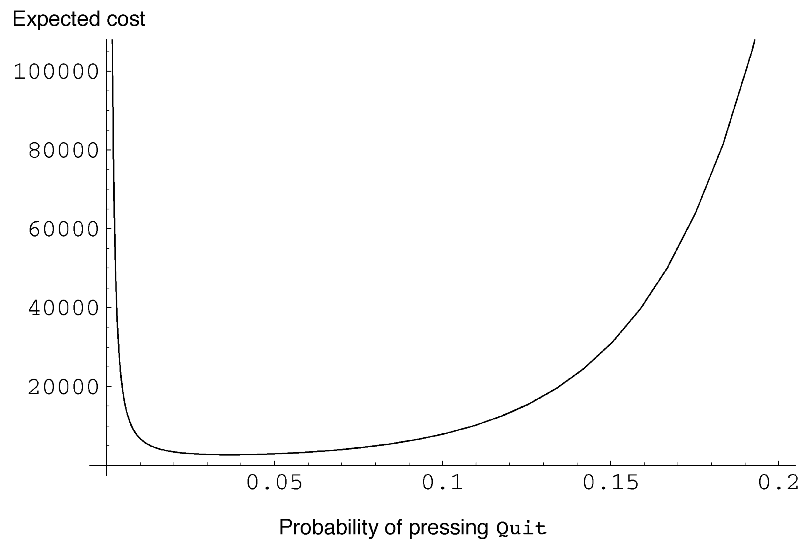

Fig. 6. Expected cost of using the Nokia 2110, depending on the probability of use of the `Quit` button (the other buttons being pressed equiprobably). This is a knowledge/usability graph, where increasing knowledge (moving toward the right in the graph) is to prefer using the `Quit` button. The graph shows clearly that there is an optimal use of the `Quit` button: if used too little (at the left of the graph), the user gets slowed down, unable to correct mistakes; if used too much (beyond the right of the graph), they get stuck, only able to correct mistakes. The graph climbs off to an impossible device as k becomes closer to 0 or 1.

very tedious—of course, if a device manufacturer collaborated with us, or if the analysis was performed by the manufacturer, the device specification should be known explicitly. Finding the transition matrix would then be completely trivial.

3.5 The *Genius* Microwave Cooker

The *Genius* is a Panasonic microwave cooker. The digital clock can display any decimal number 00:00 to 99:99, though it can only run when it is showing a 12-hour time (between 01:00 and 12:59). There are six buttons that control the clock. There are four buttons, one below each digit, which are used to increment the corresponding digit. One button, `Reset`, resets the clock to its initial switch-on state. One button, `Clock`, makes the clock run if it is showing a valid 12-hour time. The clock can be in several modes (class of state): it can have its colon flashing or stable, and the clock may or may not be running.

Although we have included the transition to switch the clock on (e.g., as occurs when the microwave cooker is plugged in to an electricity supply), we shall ignore transitions like switching it off (presumably, unless the user gets cross with the clock, they will not switch it off)! The *Mathematica* specification of this simplified *Genius* is shown in full in Figure 7.

The clock (as specified) has 30,000 states, but it is easier to visualize it as a four-state device: (1) a start state, "switched-off"—from which the only transition is to press the `clock` button to switch the display on to show 00:00, an

```
startState = s[stable, 0, 0, 0, 0];
clock[s[stable, a_, b_, c_, d_]] =
            s[flashing, a, b, c, d];
clock[s[flashing, a_, b_, c_, d_]] :=
            s[running, a, b, c, d]
                                 /; a+b > 0 && 10a+b < 13 && 10c+d < 60;
reset[s[flashing, _, _, _, _]] =
            s[flashing, 0, 0, 0, 0];
hour10[s[flashing, a_, b_, c_, d_]] =
            s[flashing, Mod[a+1, 10], b, c, d];
hour1[s[flashing, a_, b_, c_, d_]] =
            s[flashing, a, Mod[b+1, 10], c, d];
minute10[s[flashing, a_, b_, c_, d_]] =
            s[flashing, a, b, Mod[c+1, 10], d];
minute1[s[flashing, a_, b_, c_, d_]] =
            s[flashing, a, b, c, Mod[d+1, 10]];
```

Fig. 7. The *Genius* clock in *Mathematica*. A state is represented as `s[`*`mode, tens-of-hours digit, units-hours digit, tens-of-minutes digit, units-minutes digit`*`]`. The clock starts in `startState`. The rules specify the behavior of buttons, by defining their functions on the state. Rules to stop the clock running are not shown.

invalid time; (2) a state where the clock shows a valid time but is not running; (3) a state where the clock shows an invalid time; and (4) a state where the clock shows a valid time and is running. A run through the 64,322 possible state transitions provides the following transition probability matrix:

$$P = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & {}^{17}\!/_{36} & {}^{13}\!/_{36} & {}^{1}\!/_{6} \\ 0 & {}^{7}\!/_{464} & {}^{457}\!/_{464} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

We have thereby reduced a huge matrix to a 4×4 matrix, and the sums will be much more manageable. Appendix C proves the validity of this approach.

The expected number of state transitions to get from switch-on to the clock running is 216.905.[4] If we make the obvious modification and remove state 3, so the clock can only show valid times (we also have to modify the initial state so that at switch-on the clock shows a valid time, such as 12:00), then the figure becomes 7—a big improvement.

Interestingly, our experiments with people showed that the Markov model does better than humans [Thimbleby and Witten 1993]: some humans cannot set the clock at all, because they think it is a 24-hour clock (if they are tested in the afternoon, they can easily set the *Genius* to a time after 12:59, but the clock will not run). This is an example of where knowledge makes using a device harder; conversely, it is an example where the designers' tacit knowledge ("nobody uses 24-hour clocks") was accidentally built into a device. If you think you know how something works, but you are wrong, you may be permanently stuck; the Markov analysis suggests, that under such circumstances (should you notice them), you should press buttons at random. This would then give

[4]As indicated in Appendix C, the number of button presses expected of the user will be higher than the number of state transitions.

an example of how to set some time, and having discovered that, it should be easier to set the required time.[5]

We could argue that the reason why children can use devices more easily than adults is because they do not have adult preconceptions about how devices should work; they are more like ignorant Markov models! Or, to put it another way, the reason why adults find gadgets awkward is that they are not designed properly.

3.6 Combination Locks

Most devices are supposed to be easy to use, and therefore should have low transition costs. In contrast, some devices are intended to be hard to use. We now consider a simple security lock.

Consider a dial security lock, where the user can spin a dial, and they are supposed to select the right number. If we simplify the device to a two-state machine and a dial with a chance of p of being set correctly (e.g., with $^1/_p$ choices, only one of which is correct), it has a probability matrix

$$P = \begin{pmatrix} 1-p & p \\ 0 & 1 \end{pmatrix}.$$

If the user knows nothing, the mean time to "cracking" the lock (using the formula from Appendix B) is $^1/_p$. In general, if the user knows k (defined as before), the mean time is $1/(k + p - kp)$. If p is large (i.e., the security lock is easy), then it does not really matter how much the user knows: the lock is easy to open even if the user does not know the right combination. If p is very small, then the usability/knowledge curve is a hyperbola, $^1/_k$, which means, roughly, the less you know the more a little knowledge helps.

4. FURTHER WORK

Markov modeling provides a robust and general-purpose tool for user interface design and analysis. There are therefore many opportunities for further work. Here we list just a few research areas that are opened up:

—Although *Mathematica* provides both simulation and analysis, it is a general-purpose mathematics package rather than a "designer-friendly" design tool! With more work, conventional design tools could be extended to provide analysis of user interface designs. All technical details of the analysis could be hidden, and designers could be presented with estimates or graphs of usability or task times. Of course, such tools could also simulate the behavior of systems with actual use (as we did in Figure 2), and could therefore calibrate their analyses from empirical data. Developing such a tool (Thimbleby and Addison [1996] is an example) that is both powerful enough to be worth using in product design and that is nonetheless easy enough to use by practitioners would be a significant project.

—Many definitions of usability are couched in terms such as "a percentage of users can complete a percentage of tasks in a given time." Such definitions,

[5]Not that microwave cookers need to know the *correct* time anyway!

with the appropriate calibration, are answerable by using Markov models. The development of a suite of mathematical techniques to work with this conceptualization of usability should be straightforward. (Once done the details of the mathematics could be hidden from designers, by being encapsulated inside design tools.)

—In Appendix C we briefly discuss how to coalesce states in an FSM to provide alternative models of a system. Given that system designers and users typically have different models of a system, further work here would be very productive. For example, a system engineer never represents a large FSM of millions of states, but instead uses a programming language with conditionals, rules, and functions. Can suitable user interface languages guide designers into creating structures that are more easily modeled by users? For example Statecharts are one possibility: they are widely recognized as powerful design notations, and coincidentally they share some of the criteria discussed in Appendix C. Nevertheless we are unaware of any substantial work on whether "neat" Statecharts (or other notations, such as CSP [Magee and Kramer 1999]) would result in better user interfaces.

—Our knowledge/usability graph was based on a scalar parameter k that, ranging over 0 to 1, covers random to perfect knowledge of a device. We did not consider users who may have mistaken knowledge, and whose actions may be counterproductive rather than just inefficient. Since there are many ways to have incorrect knowledge, merely allowing k to be negative, say, is inadequate. A user may correctly know how to achieve the wrong goal, by confusion of function names. Generalizing our approach to knowledge, to include conceptual errors, but not losing sight of the focus on usability and of obtaining design insights, will be a difficult, and perhaps fruitful, line of research.

—Push-button devices are an easy target for FSM representation because they can be understood as a set of states for which button presses are state transitions. However, by not constraining states of the model to internal states of the device, it is possible to examine other systems. For example, if states were equated with tasks and goals, different FSMs could represent the same system as seen by a variety of users from novices through to experts. The available transitions would represent the knowledge and purposes of the user. Thus more realistic models of users would permit analysis of termination errors and other cognitively motivated effects. (One way to do this would be to compose system and user models together.) Although more research is required to do this effectively, the benefits are clear: more accurate and reliable analyses, and the same or better visualization from knowledge/usability graphs.

—Once an analytic tool has been constructed it could explore the design space to search for better designs. (Genetic algorithms, or other heuristics, could be used.) A designer could specify the basic requirements, and come back later to select from a collection of selected designs found from the initial, automated search. A design tool like this might be left running for long periods searching for improvements.

—We claim Markov models provide useful insights into important design issues, but without further work, we do not know how professional designers might be influenced by the approach. It may be that user interface development teams would not be happy with mathematical approaches, and would prefer (and be better skilled at) empirical methods.

5. CONCLUSIONS

This paper suggested a user interface design analysis method, based on Markov models, with the advantages of mathematical clarity, ease of simulation, and which provides numerical measures (and visualizations) that are ideal for comparing designs. The paper gave complete details of the approach, and gave worked examples based on commercial products.

The approach is fully operational; the Appendices give both the mathematical foundations and the program code that can be run and used directly. The approach can be applied to abstract designs, prototypes and animations, or to fully working systems. It is scalable and can accommodate complex systems, easily on the scale of typical interactive devices. It is applicable throughout the design cycle, at early design stages or late; the approach can also be used for conformance testing.

Results are conservative and do not rely on psychological assumptions, though the approach can accommodate empirical data when this is available. Certainly, the approach rather obviously lacks realism in the "look and feel" of the user interface, but this is not a disadvantage because it means the approach can be used well before designers have settled a design's look-and-feel issues.

Finally, it will be preferable to embed approaches such as ours inside design tools to provide designers with the power of the methods without the dependency on craft knowledge. In principle this is easy to do, and would bring many advantages. Design insights (of the sort described in this paper) would be accessible to professional designers at the earliest stages of design work, where they would have a critical impact on the design process.

APPENDIX

A. COMPLETE *Mathematica* CODE

The *Mathematica* code shown in this appendix is *complete working code* to simulate a user interface and to draw the figures and calculate the numbers quoted in the body of the paper. For concreteness, this appendix uses the definition of the microwave cooker from Figure 1. By editing the definition other devices can be simulated and analyzed directly.

The code starts by loading the standard *Mathematica* package for combinatorics (to load a shortest-path function, which we will need for calculating the designer's optimal transition matrix), and defines a utility routine.

```
<<DiscreteMath`Combinatorica`;
IndexOf[vector_, e_] := Position[vector, e][[1, 1]];
```

Here is Jonathan Sharp's definition of the device, written in *Mathematica* notation:

```
device =
  { {"Clock", "Clock", "Clock", "Clock", "Clock", "Clock"},
   {"Quick Defrost", "Quick Defrost", "Quick Defrost",
    "Quick Defrost", "Quick Defrost", "Quick Defrost"},
   {"Timer1", "Timer1", "Timer2", "Timer1", "Timer2", "Timer1"},
   {"Clock", "Clock", "Clock", "Clock", "Clock", "Clock"},
   {"Clock", "Quick Defrost", "Power1",
    "Power2", "Power1", "Power2"}
  };
```

This defines the device table, with rows corresponding to buttons and columns corresponding to states. We need to define the button names (which coincidentally have almost the same names as states) and define the order of the columns:

```
buttonNames = {"Clock", "Quick Defrost", "Time", "Clear", "Power"};
stateNames  = {"Clock", "Quick Defrost", "Timer1",
                "Timer2", "Power1", "Power2"};

numberOfStates = Length@stateNames;
numberOfButtons = Length@buttonNames;
```

Mathematica itself has powerful typographical features that could be used to define and present the definition of `device` exactly as in Figure 1.

A.1 Example Analysis and Graph Drawing

The first analysis discussed in the paper was for tasks getting from state `Power1` to `Power2`.

```
start = IndexOf[stateNames, "Power1"];
goal  = IndexOf[stateNames, "Power2"];
```

The random user matrix (called P in the paper) is directly calculated from `device`; button presses are treated as equiprobable, contributing `1/numberOfButtons`:

```
randomUser = Table[0, {numberOfStates}, {numberOfStates}];
Do[randomUser[[i, IndexOf[stateNames, device[[b, i]]]]]
              += 1/numberOfButtons,
  {b, numberOfButtons}, {i, numberOfStates}];
```

The designer's matrix `designer` (called D in the paper)[6] is based on the optimal route from the start to the goal states. Notice how the random user matrix (which has nonzero elements precisely where there are transitions) is converted to a `Graph` type to find shortest paths. The definition of `designer` depends on the choice of start and goal states.

```
designer = Table[0, {numberOfStates}, {numberOfStates}];
Do[Module[{p = ShortestPath[Graph[randomUser, {}], i, goal]},
             designer[[i, If[ Length[p] > 1, p[[2]], i]]] = 1
          ],
   {i, numberOfStates}];
```

[6]In *Mathematica* `D` is normally a derivative operator.

After defining the identity matrix of suitable dimensions and some utilities, we can give the definition of the mean first passage time in direct form (the formula used is derived in Appendix B):

```
ZeroRowCol[matrix_, rc_] :=
  Table[If[ i == rc || j == rc, 0, matrix[[i, j]]],
       {i, Length@matrix}, {j, Length@matrix}];

meanFirstPassage[matrix_, start_, start_] := 0;

meanFirstPassage[matrix_, start_, goal_] :=
  Module[{One = Table[1, {Length@matrix}],
          Id = IdentityMatrix[Length@matrix]
         },
      (Inverse[Id-ZeroRowCol[matrix, goal]] . One)[[start]]
   ];
```

The expected time to get from the start state (`Power1`) to the goal state (`Power2`) is `meanFirstPassage[randomUser, start, goal]`, which equals 120. The knowledge/usability graph can be plotted by `Plot[meanFirstPassage[k designer + (1-k)randomUser, start, goal], {k, 0, 1}]` (see Figure 4).

A.2 Simulating the User Interface

To simulate the device a global variable keeps track of the state of the device as buttons are pressed. We start the device in the initial state `Clock`, by writing `state = "Clock"`.

The following code constructs a row of buttons to control the device directly from the device specification, thus ensuring mathematical and empirical analysis are consistent.

```
CellPrint[Cell[BoxData[RowBox[
  Map[ButtonBox[#,
                ButtonFunction:>press[#],
                ButtonEvaluator->Automatic
                ButtonFrame->"DialogBox"]&,
      buttonNames]]], Active->True,
      TextAlignment->Center, FontFamily->"Helvetica", FontSize->24]];
```

The result is a row of working buttons (the final line of code provides the fonts and sizes as used in Figure 2). When a button is pressed, the function `press` is called as the action. A basic definition of `press` is given below, simply showing the name of the current state in the display, though it is possible to display any image if required.

```
press[theButton_] :=
  Module[{nb = ButtonNotebook[]},
    state = device[[IndexOf[buttonNames,theButton],
                    IndexOf[stateNames,state]]];
    NotebookFind[nb, "display", All, CellTags];
    SelectionMove[nb, All, CellContents];
    NotebookWrite[nb, Cell[state]]
  ];
```

The device's simulated display is a cell with name `"display"` so the function `press` can locate it: `Cell["", CellTags -> "display"]`, which would be displayed as in Figure 2. The final result can be framed, colored, formatted, and positioned to suit aesthetic requirements.

B. MARKOV MODELS AND EXPECTED FIRST ENTRY TIME

This appendix provides details of how a Markov chain can be used to model a user interface. Using this model, we extract the average times (numbers of steps, button presses, or state transitions) required to get from one state of the user interface to another.

Although many user-relevant measures can be obtained from Markov models, in the body of the paper we only used the mean first passage time M_{ij}, the expected number of state transitions to first reach a state j starting from state i. Specifically, for a transition probability matrix P, M is the corresponding matrix of mean first passage times:

$$M = \left\{ M_{ij} \,\middle|\, i = j : 0;\; i \neq j : M_{ij} = ((\mathrm{I} - [j{\downarrow}P])^{-1} \cdot \mathbf{1})_i \right\}$$

where $[j{\downarrow}P]$ is the submatrix of P with row j and column j set to zero,[7] $\mathbf{1}$ is a vector of ones, and I the identity matrix. Writing the formula out in words:

—for $i = j$, the mean first passage time M_{ij} is zero;

—for $i \neq j$, given the matrix P, set row j, and column j to zero, subtract from the identity matrix and find the inverse. The sum of row i of this inverse defines the value M_{ij}.

The mathematics may look complicated, but it provides a solid formal basis for the metrics used in this paper. Moreover the mathematics need only be done once to cover a very wide range of different systems: simply define the matrix P and initial and final states as appropriate. In practice, the analysis would be "inside" a user interface development tool, and a designer need not be concerned with the formulae themselves. *Mathematica* can solve such equations either numerically (e.g., to plot graphs) or symbolically, for example using the code given in full in Appendix A. As an example, the matrix P_{Ring}, discussed in the body of the paper, gives an expected first entry time for one state of $\frac{5-16p+21p^2-12p^3+3p^4}{6(1-4p+7p^2-6p^3+3p^4)}$. The discussion of the combination lock (Section 3.6) made simple use of such symbolic solutions.

The rest of this appendix attends to the formal derivation of the formula.

The elements of probability theory required to describe this model may be found in Grimmett and Stirzaker [1992]. For events A and B we use $\Pr(A)$ to denote the probability of A occurring, and $\Pr(A \mid B)$ to be the probability of A occurring given that B has occurred. For a random variable X, $\mathrm{E}(X)$ denotes the expectation or the mean value of X.

Assume the set of all possible states of a user interface is S, and for simplicity assume that $S = \{1, 2, \ldots, |S|\}$. The Markov chain for the user interface is the sequence $\{X_n \mid n = 0, 1, \ldots\}$ where each X_n is a random variable that takes values in S. Define, for all $i, j \in S$, the one-step transition probability to be

[7] $[j{\downarrow}P]$ is a transition matrix where state j is neither reachable nor has any action.

$P_{ij} = \Pr(X_1 = j \mid X_0 = i)$. Indeed, as the user interface does not change with time, we have that

$$P_{ij} = \Pr(X_{n+1} = j \mid X_n = i) \text{ for all } i, j \in S, n = 0, 1, \ldots$$

As it is certain X_{n+1} has some value regardless of the value of X_n, we have

$$\sum_{j \in S} P_{ij} = 1. \tag{1}$$

As in the paper, take $P = \{P_{ij} \mid i, j \in S\}$ to be the matrix of all the one-step transition probabilities. In other words, Eq. (1) is the already familiar result that the rows of the probability transition matrix P sum to 1.

For each $j \in S$, define $N(j) \mid X_i = k$ to be the random variable being the number of steps to reach state j starting from $X_i = k$. Then take M_{ij} to be the mean number of steps to reach j given that the system started in state i. That is,

$$M_{ij} = \mathrm{E}(N(j) \mid X_0 = i).$$

Trivially $M_{ij} = 0$ for $i = j$; so from now on take $i \neq j$. Now, by standard results on conditional expectations,

$$M_{ij} = \sum_{k \in S} \Pr(X_1 = k \mid X_0 = i) \times \mathrm{E}(N(j) \mid X_1 = k),$$

i.e., we have the equation

$$M_{ij} = \sum_{k \in S} P_{ik} \times \mathrm{E}(N(j) \mid X_1 = k). \tag{2}$$

Now if $X_1 = j$ then the process stops as the system has reached j. Thus, $N(j) = 1$ and $\mathrm{E}(N(j) \mid X_1 = j) = 1$. However, if X_1 is anything other than j, which we may write $X_1 = k$ for $k \in S \backslash \{j\}$, then the process continues as if it had started from k. Hence $\mathrm{E}(N(j) \mid X_1 = k) = \mathrm{E}(N(j) + 1 \mid X_0 = k)$. That is, for $k \neq j$

$$\begin{aligned} \mathrm{E}(N(j) \mid X_1 = k) &= \mathrm{E}(N(j) \mid X_0 = k) + 1 \\ &= M_{kj} + 1. \end{aligned}$$

Thus Eq. (2) becomes

$$\begin{aligned} M_{ij} &= P_{ij} + \sum_{k \in S \backslash \{j\}} P_{ik} \times (M_{kj} + 1) \\ &= \sum_{k \in S \backslash \{j\}} P_{ik} \times M_{kj} + \sum_{k \in S} P_{ik} \end{aligned}$$

which from Eq. (1) gives us, that for all $i, j \in S$ such that $i \neq j$,

$$M_{ij} = 1 + \sum_{k \in S} P_{ik} \times M_{kj}.$$

This system of equations can be simplified. Let M_j be the column vector of the M_{ij} for which $i \neq j$. The equations are now given by

THEOREM B.1. $M_j = \mathbf{1} + [j{\downarrow}P] \cdot M_j$.

where **1** is the column vector of the appropriate size made up entirely of ones.

To use Theorem B.1 to find the average time to get from one state to another, we need to be sure that by solving the equation we actually have found the M_{ij}. Fortunately, we have

THEOREM B.2. *For each $j \in S$, $(\mathrm{I} - [j{\downarrow}P])^{-1}$ exists.*

It therefore follows that

COROLLARY B.3. *The system of equations given in Theorem B.1 has a unique solution given by*

$$M_j = (\mathrm{I} - [j{\downarrow}P])^{-1} \cdot \mathbf{1}.$$

Hence, as required

$$M_{ij} = ((\mathrm{I} - [j{\downarrow}P])^{-1} \cdot \mathbf{1})_i.$$

The proof of Theorem B.2 is a consequence of the following two lemmas.

LEMMA B.4. *For each $k \in S$, $[k{\downarrow}P]^n \to \mathbf{0}$ as $n \to \infty$. That is, $[k{\downarrow}P]^n_{ij} \to 0$ as $n \to \infty$.*

LEMMA B.5. *If for some square matrix of real or complex numbers, A, $A^n \to \mathbf{0}$ as $n \to \infty$ then $(\mathrm{I} - A)^{-1}$ exists and*

$$(\mathrm{I} - A)^{-1} = \sum_{n=0}^{\infty} A^n.$$

This last lemma is well known (see Seneta [1981]), so we omit the proof. However, an informal check will easily show that the definition given of $(\mathrm{I} - A)^{-1}$ satisfies, as it should,

$$(\mathrm{I} - A) \cdot (\mathrm{I} - A)^{-1} = \mathrm{I} = (\mathrm{I} - A)^{-1} \cdot (\mathrm{I} - A).$$

The proof of Lemma B.4 is provided for completeness as it is often given in much greater generality, and hence with complexity superfluous to our purposes (see Seneta [1981]).

PROOF OF LEMMA B.4. Fix some $k \in S$, and let $Q = [k{\downarrow}P]$. Because Q deals only with transitions omitting k, it represents the probability of getting from state i to state j in exactly one step without passing through state k. Also, Q^n is the probability of getting from state i to state j in exactly n steps without ever having passed through state k.

For a sensible user interface, it must be possible to get from any state $i \neq k$ to k after say some N steps (otherwise, the user interface has states which are not accessible!). But as Q^N represents the probability of not visiting k after N steps, it must be that

$$\sum_{j \in S \setminus \{k\}} Q^N_{ij} < 1.$$

As in this sum, but for clarity, we will implicitly take all further sums over $S \setminus \{k\}$. Now, by definition, for any n,

$$\sum_j Q^{(n+1)}_{ij} = \sum_j \sum_r Q^n_{ir} \times P_{rj}$$

which by Eq. (1) gives

$$\sum_j Q_{ij}^{(n+1)} \le \sum_r Q_{ir}^{n}.$$

Hence, for all $n \ge N$, if $\theta = \sum_j Q_{ij}^{N}$, then

$$\sum_j Q_{ij}^{n} \le \theta < 1.$$

Moreover for each $m \ge 1$,

$$\begin{aligned}\sum_j Q_{ij}^{N(m+1)} &= \sum_j Q_{ij}^{N} \sum_r Q_{ir}^{mN}\\ &\le \theta \sum_r Q_{ir}^{mN}.\end{aligned}$$

Therefore, proceeding by induction, we have

$$\sum_j Q_{ij}^{mN} \le \theta^{m}. \tag{3}$$

But as $m \to \infty, \theta^m \to 0$. Thus the sums on the left-hand side of Eq. (3) tend to zero and form a convergent subsequence of the monotone decreasing sequence of the sums

$$\sum_j Q_{ij}^{n}.$$

Thus these sums tend to 0 as $n \to \infty$. From this it follows that $Q_{ij}^{n} \to 0$ as $n \to \infty$, as each of these is nonnegative. This proves the elementwise convergence stated in the lemma. □

C. COALESCING STATES

We may wish to reduce a large set of states to a smaller number, for instance to define an alternative model. This appendix discusses the conditions under which this simplification may be performed: states can be grouped together (while preserving certain properties) provided they share next-state transition probabilities. This requirement is similar to the grouping property of Statecharts [Harel 1988] that states nontrivially grouped together share next-state transitions.

Given a probability transition matrix P, we wish to find a smaller matrix with equivalent properties, so we can use smaller Markov models and obtain the same results. We shall do this by merging two sets of states together into a single state, and call the new probability transition matrix P^*.

What are the conditions these merged states must satisfy, and what is the appropriate way of calculating P^* from P?

Since the allocation of the N state numbers is arbitrary, we can exchange rows and columns in P so that the states to be considered for merging are adjacent. If the respective state occupancy probabilities of the two states are a and b, then the complete state probability vector can be written $(a \;\; b \;\; C)$, where C is a vector.

Now let $(a \;\; b \;\; C) \cdot P = (a' \;\; b' \;\; C')$. Since the probabilities are independent, we require

$$(a + b \;\; C) \cdot P^* = (a' + b' \;\; C').$$

Write out P and P^* as conformant block matrices (so $p_{31},\ p_{32},\ p^*_{21}$ are column vectors; $p_{13},\ p_{23},\ p^*_{12}$ are row vectors; $p_{33},\ p^*_{22}$ are square matrices; and the rest are 1×1 matrices):

$$P = \begin{pmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & p_{33} \end{pmatrix}, \quad P^* = \begin{pmatrix} p^*_{11} & p^*_{12} \\ p^*_{21} & p^*_{22} \end{pmatrix}$$

Multiplying out we obtain constraints:

$$\begin{aligned} p^*_{22} &= p_{33} \\ p^*_{21} &= p_{31} + p_{32} \\ p^*_{12} &= p_{13} = p_{23} \\ (a+b)p^*_{11} &= ap_{11} + bp_{21} + ap_{12} + bp_{22} \end{aligned}$$

The last constraint must hold for all a and b, so $p^*_{11} = p_{11} + p_{12} = p_{21} + p_{22}$. This appears to be a tiresome constraint: but since P is a probability matrix (all rows add to one), the condition is equivalent to the existing constraint $p_{13} = p_{23}$. Collecting the equations we have

$$P^* = \begin{pmatrix} p_{11} + p_{12} & p_{13} \\ p_{31} + p_{32} & p_{33} \end{pmatrix}, \quad \text{provided } p_{13} = p_{23}.$$

If either a or b is always 0, then the constraints are trivial. In other words if there are states that are not initially occupied and are never reachable (which $p_{31} = 0$ or $p_{32} = 0$ would imply), then they can be deleted.

The process of pairwise grouping can be repeated, reducing any set of suitable states into a single superstate (and there is no need to first permute the states to start at 1 or be adjacent). In words, we can group any states into a single state provided only that all states in the group share identical transition probabilities *out* of the group—this is the $p_{13} = p_{23}$ constraint.

It may be more convenient to deal with frequencies rather than probabilities. For example, a system specification will typically give explicit state-to-state transitions, and "summing over states" will then directly give counts of all out-transitions for all states under consideration. Since probabilities are relative counts, a reduced matrix can be obtained by adding the appropriate counts together and dividing through each row by its total.

The transformation of P to P^* preserves the probabilities and the results obtained from a Markov model, but with one proviso: since states have been merged, the Markov model no longer "counts" transitions between those original states.

ACKNOWLEDGMENTS

The referees made substantial comments that have helped the presentation of this paper. The authors are indebted to Ian H. Witten (Waikato University) and Saul Greenberg (Calgary University) whose microwave cooker started it all. The present paper owes much to the collaborative work with Ian Witten described in Thimbleby and Witten [1993], though the present paper corrects the mathematics. Richard Young and Ann Blandford provided invaluable and extensive criticism.

REFERENCES

BALDI, M., MACII, A., MACII, E., AND PONCIO, M. 1999. Application of Symbolic FSM Markovian Analysis to Protocol Verification, *IEE Proceedings on Computers and Digital Techniques 146*, 5, 221–226.

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

CARD, S. K., PIROLLI, P., AND MACKINLAY, J. D. 1994. The Cost-of-Knowledge Characteristic Function: Display Evaluation for Direct-Walk Dynamic Information Visualizations, *Proceedings CHI'94*, ACM Conference on Human Factors in Computing Systems, Boston, 238–244.

COOK, J. E. AND WOLF, A. L. 1998. Discovery Models of Software Processes from Event-based Data, *ACM Transactions on Software Engineering and Methodology 7*, 3, 215–249.

DEAVOURS, D. D. AND SANDERS, W. H. 1998. On-the-fly Solution Techniques for Stochastic Petri Nets and Extensions, *IEEE Transactions on Software Engineering 24*, 10, 889–902.

FEYNMAN, R. P. 1996. *Feynman Lectures on Computation*, edited by J. G. Hey and R. W. Allen, Addison-Wesley.

GRIMMETT, G. R. AND STIRZAKER, D. R. 1992. *Probability and Random Processes*, 2nd ed., OUP.

HALBWACHS, N. 1993. *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers.

HAREL, D. 1988. On Visual Formalisms, *Communications of the ACM 31*, 5, 514–530.

HAREL, D. AND POLITI, M. 1998. *Modeling Reactive Systems with Statecharts: The Statemate Approach*, McGraw-Hill.

JONES, M. AND WOODLAND, P. C. 1994. Modelling Syllable Characteristics to Improve a Large Vocabulary Continous Speech Recogniser. *Proceedings ICSLP'94*, 2171–2714, Yokohama, Japan.

KNUTH, D. E. 1998. *The Art of Computer Programming 2*, 3rd ed., Addison Wesley Longman.

KIERAS, D. AND POLSON, P. G. 1985. An Approach to the Formal Analysis of User Complexity, *International Journal of Man-Machine Studies 22*, 4, 365–394.

LEWIS, H. R. AND PAPADIMITRIOU, C. H. 1998. *Elements of The Theory of Computation*, 2nd ed., Prentice-Hall International.

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

MILLER, B. P., FREDRIKSEN, L., AND SO, B. 1990. An Empirical Study of the Reliability of Unix Utilities, *Communications of the ACM 33*, 12, 32–44.

MONK, A. F. AND CURRY, M. B. 1994. Discount Modelling with Action Simulator, Proceedings BCS Conference on HCI, HCI'94, *People and Computers*, IX, edited by G. Cockton, S. W. Draper and G. R. S. Weir, 327–338, Cambridge University Press.

NEWMAN, W. M. AND LAMMING, M. G. 1995. *Interactive System Design*, Addison-Wesley.

PALANQUE, P. AND PATERNÒ, F., EDS., 1998. *Formal Methods in Human-Computer Interaction*, Springer-Verlag.

SENETA, E. 1981. *Non-negative Matrices and Markov Chains*, 2nd ed., Springer-Verlag.

SHARP, J. 1998. *Interaction Design for Electronic Products using Virtual Simulations*, Ph.D. thesis, Brunel University.

SHNEIDERMAN, B. 1998. *Designing The User Interface*, 3rd ed., Addison-Wesley.

SILFVERBERG, M., MACKENZIE, I. S., AND KORHONEN, P. 2000. Predicting Text Entry Speed on Mobile Phones, *Proceedings ACM CHI'2000*, 9–16.

STANTON, N. A. AND YOUNG, M. S. 1999. What Price Ergonomics? *Nature 399*, 6733, 197–198.

THIMBLEBY, H. W. 1992. The Frustrations of a Pushbutton World, *1993 Encyclopædia Britannica Yearbook of Science and Future Technology*, 202–219, Encyclopædia Britannica Inc.

Thimbleby, H. W. and Witten, I. H. 1993. User Modelling as Machine Indentification: New Design Methods for HCI, in D. Hix and R. Hartson, Eds., *Advances in Human Computer Interaction IV*, 58–86, Ablex.

Thimbleby, H. W. 1994. Formulating Usability, *ACM SIGCHI Bulletin 26*, 2, 59–64.

Thimbleby, H. W. and Addison, M. A. 1996. Intelligent Adaptive Assistance and Its Automatic Generation, *Interacting with Computers 8*, 1, 51–68.

Thimbleby, H. W. 1996. Internet, Discourse and Interaction Potential, in L. K. Yong, L. Herman, Y. K. Leung, and J. Moyes, Eds., *First Asia Pacific Conference on Human Computer Interaction*, Singapore, 3–18.

Thimbleby, H. W. 1997. Design for a Fax, *Personal Technologies 1*, 2, 101–117.

Thimbleby, H. W. 1999. Specification-led Design, *Personal Technologies 2*, 4, 241–254.

Thimbleby, H. 2000. Analysis and Simulation of User Interfaces, Proceedings of BCS Conference on Human Computer Interaction, in press.

Whittaker, J. A. and Poore, J. H. 1993. Markov Analysis of Software Specifications, *ACM Transactions on Software Engineering and Methodology 2*, 1, 93–106.

Wolfram, S. 1996. *The Mathematica Book*, 3rd ed., Addison-Wesley.

Wasserman, A. I. 1985. Extending State Transition Diagrams for the Specification of Human-Computer Interaction, *IEEE Transactions on Software Engineering 11*, 8, 699–713.

Young, S. J. 1993. *The HTK Hidden Markov Model Toolkit: Design and Philosophy*, Cambridge University Engineering Department Technical Report CUED/F-INFENG/TR.152.

Received August 1998; accepted August 2000