User modeling as machine identification: new design methods for HCI

Harold Thimbleby[†]
Ian H. Witten[‡]

This chapter[1] takes a formal approach to the design of a class of user interfaces, and shows that some “psychological” design guidelines can be approached without assumptions of human psychology. We suggest an operational approach to assessing user interfaces, with a theoretical foundation in systems theory. Although more work in applying formal methods to HCI is required, we provide arguments for its importance, not least because of the intrinsic unreliability of empirical methods in the face of the complexity of even the simplest of interactive systems. We also argue that much of this complexity is in any case spurious and exacerbates user interface problems, indeed, drives the debate about quality in user interfaces away from fundamental issues towards cosmetic concealing of design faults.

A reader of this chapter commented that, “I would not argue that these problems aren’t important, but they seem minor compared to the real task of designing a system that suggests and supports appropriate, task-relevant actions.” This is based on a misconception: the real task of design includes more than just making a system suggest and support task-relevant actions (and so forth). It must do so reliably, completely, and consistently (to chosen criteria): in short, it must be trustworthy. Without a formal basis — which is what we argue for — interactive systems merely give the temporary semblance of being easy to use. The thorough study of even a simple interactive system is beyond either users or conventional empirical design methods: formal analysis is essential.

Symptoms of the problem: two examples

We motivate this chapter with two case studies taken from two different interactive systems designs. For the first we compare two versions of a graphical operating system: we suggest that the changes between the versions indicate that the designers recognised the problems we discuss, but that their solutions are cosmetic. A more formal approach to design would have resulted in deeper solutions. The second example is more baffling: a microwave cooker whose user interface design has been unchanged for several years. We find that, again, the manufacturers recognise the interface problem we identify — but their approach is not to change the interface itself but to change user manuals. This is more clearly a purely cosmetic solution.

We know that design is difficult (we discuss this more fully below); the case studies we give here suggest that that poor initial design is not fixed at a later stage. We subsequently show that many problems are anyway too subtle to be detected by practical experiments (hence iterative design is insufficient). It is also possible that designers lack

[†] Department of Computing Science, Stirling University, Stirling, FK9 4LA, Scotland. Email: harold@uk.ac.stirling.compsci.

[‡] Department of Department of Computer Science, University of Waikato, Private Bag 3105, Hamilton, New Zealand. Email: ihw@waikato.ac.nz.

[1] To appear in *Advances in HCI IV*, edited by R. Hartson and D. Hix, Ablex, 1992.

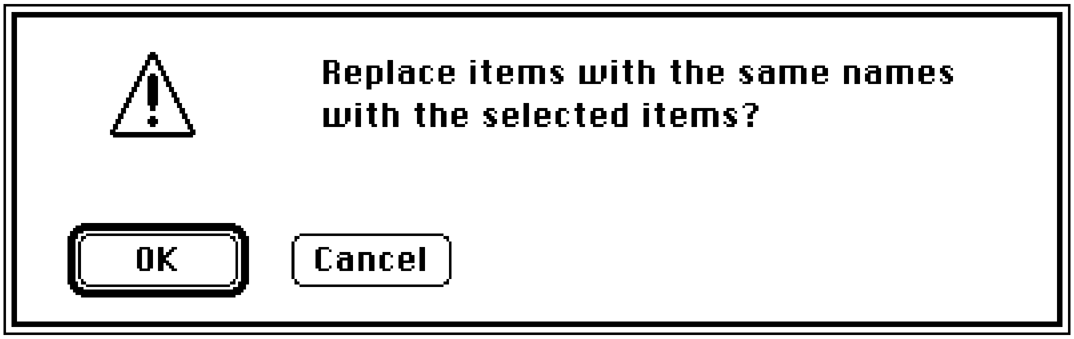
imagination or the power to influence programmers, or even that the attractive appearance of systems detracts from objective analysis. If formal methods are used early in the design process more potential user interface problems are identified and can be solved before it is too late; one obtains useful analytical insights without the prohibitive time and unreliability of empirical investigations. Indeed, empirical methods often require a prototype system, and by time one is available there has already been investment in programming and it is getting too late to fix structural — formal — problems.

Example 1: Workstation interface

Apple Computer have recently released a new operating system for the Macintosh Computer, System 7. This has been developed over the period 1983 to 1991, and since Apple are major proponents of user interface design, we can assume that the best methods have been used — certainly the operating system has been iterated through previous versions, both released and internal to Apple. There is also a large literature on Apple's user interface work and their recommendations (e.g., Apple, 1987).

Nevertheless, serious flaws remain. Let us take a simple example.

In System 6, it was sometimes the case that copying a file from one folder to another would result in a name clash. The system then displayed the dialog box shown in Figure 1.

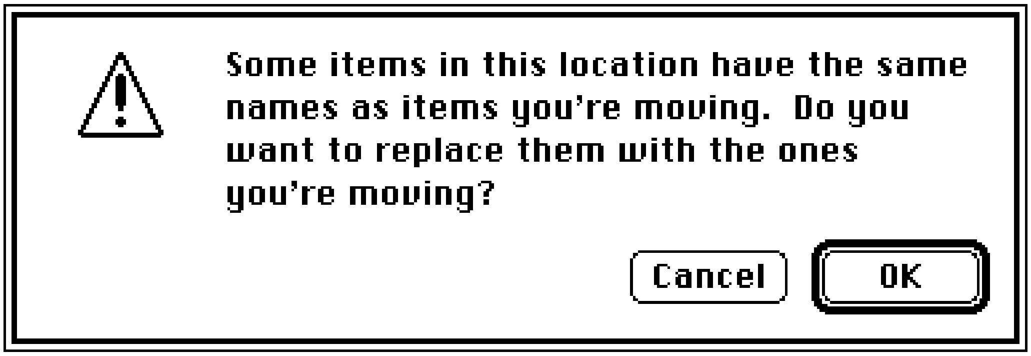


Replace items with the same names
with the selected items?

Figure 1. System 6 dialog box

This is difficult to understand, and does not give the user a computational handle on the problem: a name clash is symmetric, and so the copy could proceed in either direction copying either of the files concerned. For example, the user may wish to replace the older version of the file, or perhaps to recover a file from an older backup. There are other possibilities. The user is required to remember the names of the files involved, since the dialog box may conceal them.

By System 7, the designers have noticed the problem (or at least it now has sufficient priority to be tackled). The dialog box is redesigned (Figure 2).



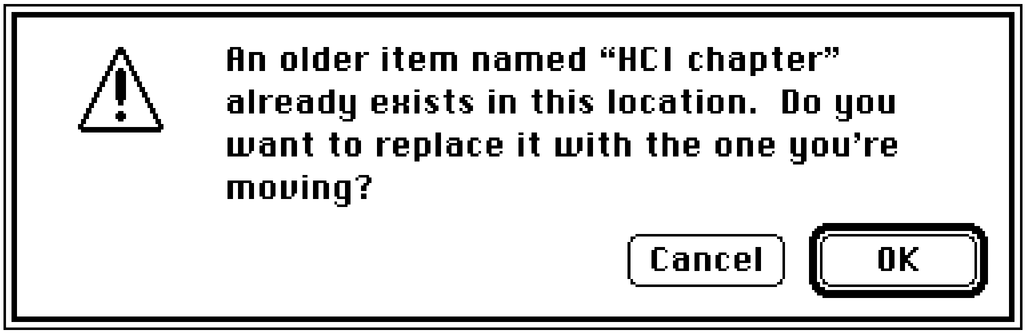
Some items in this location have the same names as items you're moving. Do you want to replace them with the ones you're moving?

Figure 2. System 7.0 dialog box

The designers have improved the obscure explanation. The referent of “this location” is not clear (the dialog box may also obscure it). The user interface functionality appears to be unchanged. However, a major difference in functionality is that now, choosing Cancel from the dialog box deselects the files concerned, while previously the set would have remained selected. Whereas the user could previously experiment by deselecting files in turn to identify the name clash more specifically, now he has to remember the selection and redo it manually. As before the dialog box is immovable, so it is not generally possible to see which files are concerned.

It appeared to us, then, as new users of System 7, that the designers had noticed a problem and addressed it by palliating the English. Of course this is useful, but it is not sufficient.

Consider our surprise when we started experimenting to determine the specific files causing the name clash. On attempting to copy a single file, we obtained Figure 3.



An older item named “HCI chapter” already exists in this location. Do you want to replace it with the one you're moving?

Figure 3: File-specific System 7.0 dialog box

This dialog box now indicates the specific name of a file, and that the target file is older. So Apple have not only noticed the problem with the English style, but also the computational problem. (However, they don’t provide the natural converse option, to replace the newer with the older file.) The word “older” or “newer,” as appropriate, is computed by comparing the files’ modification times.

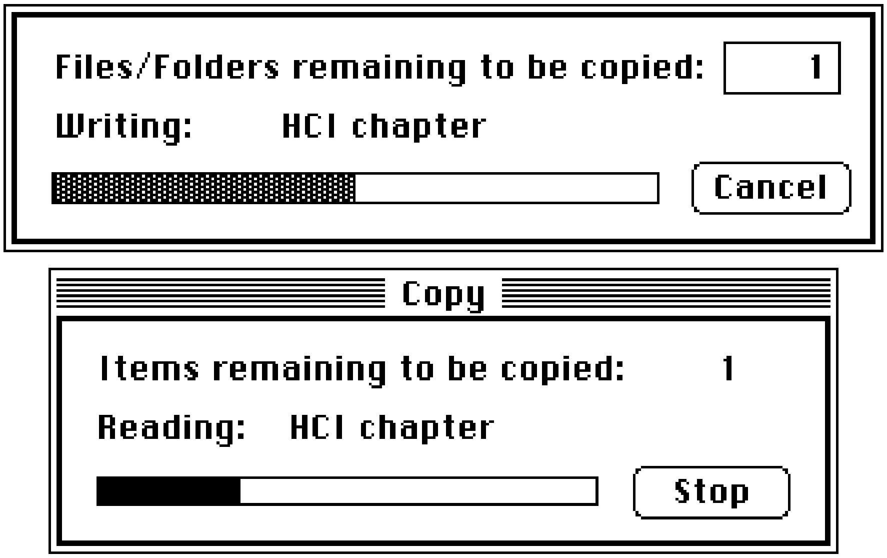
The point is that the thrust of Apple’s development, as suggested by this example, seems to be in palliating user interface design. In file age comparison they solved a special case of the computational problem: for one file. Without the dialog box (Figure 3) we could have charitably assumed that Apple were unaware of the more general design problem, but it is clear that ignorance is not the excuse. Imagine a debate between Apple and a computer scientist over this point:

> *Apple?*
> The empirically based expectation of requiring the general feature (more than one file) is small. The improvement to the English is of benefit to everyone. The dialog box design is consistent with the user interface guidelines.
>
> *Computer scientist?*
> You have recognized the case for *n*=1, why not implement an interface supporting the general case? The improvement to the English is a diversion. In what way does the new design support any algorithm the user might run to determine what files are involved? Indeed it is now computationally harder to determine the name clash, since files are gratuitously deselected. To say nothing of the bugs (see below) — surely getting the computer science *right* is more important than getting it to look *nice*?

Whatever it looks like, which we can debate the merits of, there *are* bugs. The modification times of the files are determined from caches that are not necessarily up to date. Although the user interface *looks* as though it has been improved with "helpful" English, the user is now given deceptively useful information: the dialog box may say one file is older than another when the reverse is the case.

Are we being too hard?

A bug in System 6 is that the **Cancel** button that appears later during the file copying does not cancel the operation (despite Apple's own design guidelines, Apple, 1987), since the files being replaced are deleted regardless. This problem has presumably been noticed, for in System 7 there is no **Cancel** button: it is called **Stop**! It stops the copy, certainly, but it does not cancel the operation, since stopping the copy does not leave the original files intact. Again, we see computational problems being "fixed" by cosmetic solutions.



Files/Folders remaining to be copied: 1

Writing: HCI chapter

Copy

Items remaining to be copied: 1

Reading: HCI chapter

Figure 4: Copy dialog boxes, System 6 (top), System 7 (bottom)

Certainly, interactive system design is not easy, and systems such as the Macintosh practically encourage designers to seek solutions in new representations — graphical styles, sounds, animations and so forth — rather than in the underlying meaning of the system. It is apparent that implementation problems tend not to get fixed with system upgrades; in general it is likely that poor underlying implementations encourage new *ad hoc* features as spurious "work arounds."

Since there are so very many distracting design options in systems like the Macintosh, raising issues from graphics to English, we now turn to a much simpler design problem.

Example 2: Microwave oven

On a stormy evening in late May 1991 we were faced with the problem of setting the clock on a Panasonic Dimension 4 Microwave oven (Panasonic, undated). We were in the Canadian Rockies, miles from civilization, and stranded without the manual …

Since at first we didn't get the microwave's clock working, we did not think to time how long we took, but we estimate about a quarter of an hour. Were the clock setting problems "our fault," or not? After all, our preconceptions about clock setting may be atypical.

We later implemented a simulated microwave to facilitate empirical investigation of the clock setting problems we encountered. (A brief discussion of design choice in clock *representations* can be found in Plaisant & Shneiderman, 1991.) Experiments suggested the generality of our own experience: all subjects tested found great difficulty in setting the time.Since the experiments were artificial, the frustrated subjects generally gave up after around 10 minutes. The user interface to the clock *is* badly designed.

The simulation is a HyperCard program, exhibiting the microwave's buttons we had discovered were relevant to setting the clock. We did not include several buttons, such as those labelled **froz** or **comb** (of which there are two), mostly of uncertain meaning. The buttons are on a smooth plastic panel that might have included further buttons that we did not identify.

The HyperCard simulation is driven by a single mouse and therefore does not permit the user to press more than one button at a time, though on the microwave we had tried pressing several buttons together (like **clock** and **reset**); no such combinations appeared to have any useful effect. The simulation, then, is *considerably* simpler than the real microwave clock. The microwave (less the confusing cooking buttons) has the functionality and approximate layout indicated by the HyperCard rendition in Figure 4. The functionality of the clock is summarized in Appendix A1 as a production system; Appendix A2 gives the HyperCard program.

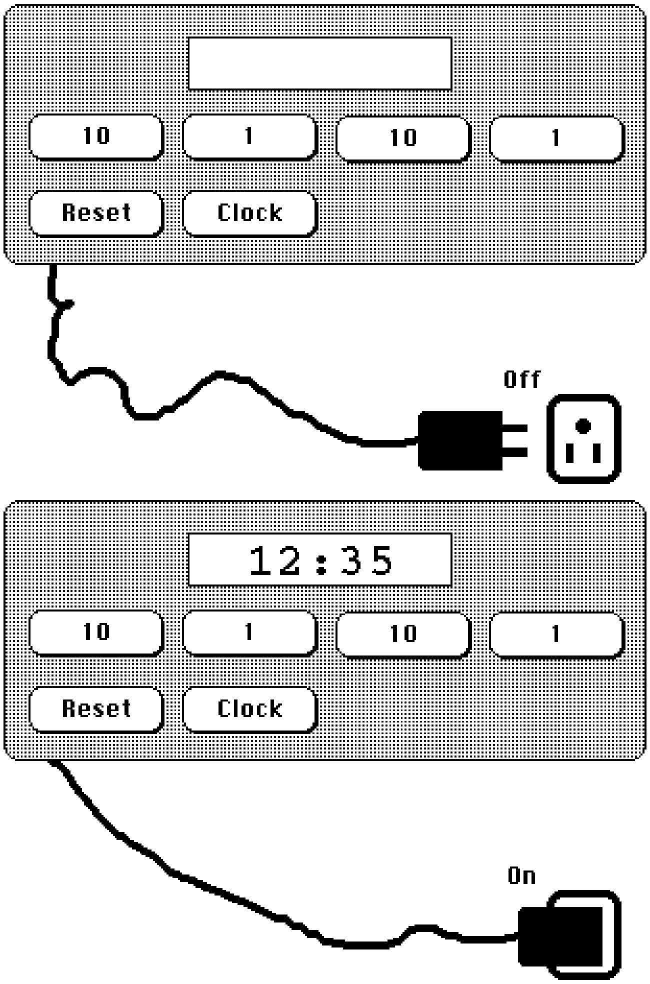


Figure 5. HyperCard representations of microwave clock setting facilities

We found users familiar with HyperCard and Macintosh basics. The task was to set the clock to the current time. The users, like us, discovered some of the following pertinent facts:

1) Plugging the machine in to the electricity supply in is necessary for it to function (the display is blank when the machine is off).
2) Unplugging the machine then plugging it in resets it. (The real microwave required a delay of several seconds after unplugging before switching it back on for the action to be effective.)
3) Pressing **reset** has no effect until **clock** has been pressed.
4) Pressing any of the top row of buttons has no effect until **clock** has been pressed.
5) Pressing **clock** the first time makes the colon flash.
6) The buttons marked **10**, **1**, **10**, **1** now independently increment the tens-of-hours, hours, tens-of-minutes and minutes digits (in that order) of the time display. Blanks (e.g., left there following **reset**) are treated like 0.
7) The four buttons increment the number displayed, not a “time.” Thus incrementing the minutes from 59 goes to 50, neither 60 nor 00; incrementing the 10 minutes from 59 would go to 69 — an invalid time. The buttons can therefore be used to set any number from 0 (shown as blank!) to 9999.

8) It is easy to set a time, such as 15:00 (which might be the time of the experiment). However the **clock** button then has no observable effect, and **reset** clears the display, leaving the colon still flashing but showing no digits.

Users are unlikely to infer or discover two important facts:

1) The problem with the design is that the clock is a twelve hour clock and can only be set when the time set is between 01:00 and 12:59. (That the clock digits can be adjusted to "times" like 15:00, and indeed to 99:99, seemed to confirm the user's model that the clock worked to a 24 hour syntax!)
2) When the clock is correctly set (and running) the colon will stop flashing; however it takes one second to see that the cursor is not flashing and about one minute to see that the clock has started running. These considerations are "theoretical" in the sense that no user had problems with them, implicitly bringing knowledge about flashing and running digital clocks. There were no empirical problems with these (unnecessary) features of the design.

We ourselves only managed to set the clock of the prototype after thinking that, since the microwave timed cooking with relative times, in principle it did not need — if only we could find out how — to have its clock set. An accident in testing this hypothesis resulted in us setting the clock to 01:00 (or some such), and hence discovering that pressing **clock** was, under such circumstances, the right way to start the clock running. Prior to this, we had become fixated on the idea that something more subtle than merely pressing **clock** was required, after all the time *was* 22:07 and the clock could *clearly* count up to 22:07 and beyond!

We were subsequently able to find a manual for the microwave:

1) We had failed to discover that when the clock is running, if the **clock** button is pressed twice within two seconds, the clock resets. (In our experiments, we had only been able to reset the clock by unplugging the microwave from the electric supply. Although we had tried "chord" inputs — pressing several buttons at once, such as **reset**+**clock** — we had not discovered any new effects with rapid pressings. This is especially surprising since we did not transfer our Macintosh "double click" experience.)
2) The manual has a small note that the clock is 12 hour, "with no distinction between AM and PM." Had the microwave been better designed, for example so that the user interface restricted the user to numbers corresponding to the 12 hour clock, it would not have been necessary to mention this problem in the manual. This supports Thimbleby's view that improving manuals is a heuristic for improving systems (Thimbleby, 1990).
3) The manual claims that "when a pad is touched correctly, a beep will be heard." Yet the **clock** button does *not* beep when it is pressed to successfully start the clock running.

More recent models of microwave cooker have a clock with the same problems. As with Apple, it is unlikely that Panasonic are unaware of the user interface problem: the newer manuals (Panasonic, 1991) mention the same twelve hour restriction, somewhat more prominently that in the manuals of earlier models. However, the systems themselves appear to be unchanged. Our motivation for this chapter, then, is to ask what sort of knowledge could the Panasonic designers have had available at the appropriate point in the design process to avoid the problem we have discussed?

Machine identification

We have given two examples of bad user interface design caused by lack of attention, or lack of priority, given to the semantics underlying the interface. What do users do with systems, and how can we capture that in a framework amenable to reasoned design at an early enough stage in the design cycle to make an impact?

Users engage in a process of *modeling* to determine a conceptual structure that corresponds to the machine being operated. The purpose is to create a predictive model that can be used to anticipate the effect of pressing any button in any state, and to enable the machine to be driven into certain desired states. Modeling involves first eliciting behavior from the machine and then interpreting the behavior sequence, for instance recognizing or identifying patterns that can be readily comprehended and used.

In order to exploit a computational perspective on design, we need a general notation that is not biased towards particular forms of expression, and for which a body of theory exists. We will use finite-state automata (abbreviated FSA) (Conway, 1971; Minsky, 1967). These are sufficient for any computer system since each bit in the data can be interpreted as a state and the program can be interpreted as a state-transition function.[2] Of course there is a difference between the FSA of the program and the FSA of the user interface; we are concerned with the latter.

Several important classes of machine can be identified:

1) ***Simple***
The internal state of the machine is a function only of the current button press(es). Most musical keyboard instruments are simple.

2) ***Declarative***
The internal state of the machine is a projection of the observable state. That is, there is a projection function from observations to states (e.g., how to map numbers and flashing colons to a state number).

3) ***k-Declarative***
The internal state of the machine is a projection of (at most) the last *k* observable states and inputs, for some fixed value of *k*.

4) ***Hidden state***
The internal state cannot always be determined from the last *k* observable states and inputs, for any fixed value of *k*. That is, the machine has modes that persist over input sequences of arbitrary length.

5) ***Non-deterministic***
The transitions that take one state to the next are non-deterministic. All deterministic machines states can be viewed as non-deterministic ones with fewer states.

   Non-deterministic machines are of particular importance when we consider user modelling. If the user has not identified all states of a machine with which they are interacting, through lack of time, misconception or for any other reasons, their model is necessarily non-deterministic.

6) ***Temporal***
The state of the machine depends on the timing of the input. Timeouts, double-clicks, etc. are examples of temporal input.

Domain knowledge and representation by NDFSA

A non-deterministic FSA (NDFSA) is one way of representing a FSA, but generally using fewer states. Each of its states represents a set of states of the FSA, and the state-transition function of a FSA becomes a relation in a NDFSA (or equivalently a mapping to the power set of states). A NDFSA can be constructed for any FSA and conversely. It follows

[2] They are not, however, a ‘good’ representation for systems more naturally modelled, for instance, by push down automata. This need not concern us here, since (i) PDA have no advantage for modelling finite behavioral data, (ii) humans are anyway supposed to have small stacks. Discussion of the relative complexities of different models can be found in (Gaines, 1977).

that the former accept the same languages as the latter, their advantage being that for some purposes they are more manageable.

The microwave clock can be represented conveniently as a 5-state NDFSA (see Figure 6). A human user more easily learns a 5-state NDFSA than the presumed 10721 state[3] FSA, if only because humans don't live long enough to learn the latter. The Figure shows the states of the NDFSA labelled with terms like "valid time": these labels are quite arbitrary but illustrate how the states of the NDFSA reflect domain knowledge, which we have explicated here by our choice of mnemonic labels.

The state transition probabilities shown in Figure 6 are computed assuming equiprobable button presses amongst those transitions we discovered were possible without the manual (!) and equiprobable state occupancy. The system is not ergodic. We later discovered that a transition is possible from Clock running to Reset by double pressing the **clock** button; we, in our ignorance, had to unplug the microwave and hence had a "transition" from Clock running to Power up.

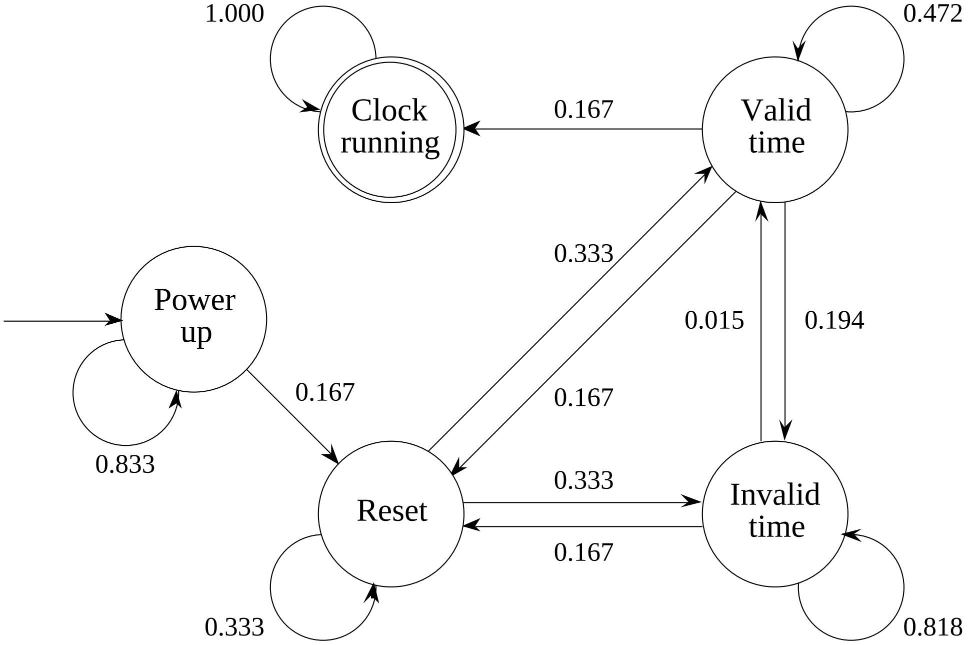


Figure 6. Microwave clock NDFSA, showing state transition probabilities

The number of *n*-state NDFSAs corresponding to an *N*-state FSA is huge even for small values of *n* and *N*. The problem of identifying non-deterministic structures with a specified number of states that provide an optimal fit to a given behavior sequence has been investigated by Gaines (1975), who found no alternative to the massively computation-intensive technique of enumerating all possible structures and calculating how well they match the sequence (see also Gaines, 1977, and Courtois, 1985, for a review). Since exhaustive techniques are prohibitively expensive, it is likely that humans use different strategies, such as the greedy method (i.e., once the user has partitioned the states into two classes, those distinctions remain, though they may be refined).

An important reason for concentrating on NDFSAs is that they can be used to capture domain knowledge:

[3] One state with colon not flashing and clock not runnable (power up state); 720 states with the colon not flashing and clock running; 10000 states with the colon flashing and clock not running.

> *Each state in a suitable non-deterministic automaton collapses all knowledge-equivalent states of the deterministic FSA. The state identification function encapsulates declarative knowledge; the state transitions encapsulate procedural knowledge. The knowledge is typically probabilistic.*

Thus the trivial NDFSA with a single state collapses all behavior to a constant, the domain knowledge, "this is a system." With two states, we can represent the domain knowledge "on" *vs* "off." And so forth; knowing the 5 state NDFSA above is sufficient to operate the microwave clock (upto the equivalence of valid times).

Two states may be used to represent any other distinction instead (e.g. on the microwave, clock running/not running; 12:03/not 12:03), and one generally chooses the states using some objective function to maximize their informativeness. In this example, on/off are perhaps equally likely and hence this interpretation (otherwise relying on domain knowledge) of the machine maximizes the information content of the two-state NDFSA.

In short, a NDFSA with suitably chosen states is a convenient representation of plausible — and effective — domain knowledge, up to isomorphism. Identifying NDFSAs (by exhaustive reduction or behavioral methods such as those of Gaines, 1975) classifies finite-state behavior by domain knowledge.

Random experiments

A random process providing inputs (button presses) to a FSA is at a great disadvantage with respect to a human user. Domain knowledge, such as the way digits work (4 is the successor of 3, and so forth) and much assumed domain knowledge (e.g., digits denote things with arithmetical properties) is a major help for the human. A serious problem for all psychological approaches to design is when to *stop* invoking domain knowledge: is knowing that "the microwave uses a 12 hour clock" appropriate domain knowledge or not? (A designer who tacitly assumes this will have no way of improving the design with respect to less specific domain knowledge.) The use of a random process gives *conservative* results that are unaffected by domain assumptions, sidestepping the domain knowledge (and psychological) problems together with the designer's temptation to cheat with situation-specific "domain knowledge". Furthermore, modelling a random process is theoretically tractable.

In contrast to random processes, humans have domain fixations. For example, in our attempts to set the microwave clock, we persisted in trying to set it to the current time, 22:07 then 22:10 … and later! Random processes do not have fixations, socio–cultural biases, or other "expectations." Results with a random process are therefore *unbiased.*

Systems are easier to learn and use if they are designed with the "right" conceptual model. There are an infinite number of nearly right and simply wrong conceptual models that the user might have, so the idea of a "right" conceptual model may give the designer a false view of system quality — a temptation avoided by using a random process.

Operationally, using a random process to explore an interactive system has much to commend it. It can be used (as it were) by a robot on a working system — such as a production microwave. Practical experiments show that this is a surprisingly effective way of exposing design flaws (Miller, Fredriksen & So, 1990), including even serious ones missed by prolonged exposure (decades) to users. It has been shown that random machines can generate sequences of sufficient complexity to explore FSAs of arbitrary size (Gaines, 1971).

There is no "prototyping" or "animation" environment, and no special specification language that the design has to be expressed in. Such schemes have the disadvantage not only of designer motivation but also of abstraction: for example, if the specification language cannot express timeouts or other concrete subtleties of the user interface, then the evaluation procedure inevitably remains silent about such things.

In some environments the development language will already support appropriate analysis (we discuss one below). In such cases mathematical investigation or automated, systematic exploration can supplant empirical investigation.

Discussion

How long does it take to set a clock?

If a user having trouble with a system asks a designer what to do, the designer will often say something like “press ~@# and …, that gets it working. It’s obvious!” The user is made to feel dumb! We can use our approach to expose the deceit in the designer’s position.

What is the minimum number of button presses (counting only the clock-setting buttons) to set a time on the microwave clock?

If we ask the designer, who knows how the microwave functions, he can give us a minimal sequence of button presses. This will be 3: **clock**, **1h**, **clock**.

Now consider a random sequence of button presses, that is, assuming no knowledge such as the designer used in deriving 3. If $p^{(n)}$ is the probability that the clock is running on button press n, then the probability $f^{(n)}$ that the clock is first set on the nth button press is given by

$$f^{(n)} = p^{(n)}\prod_{i=1}^{n-1}(1-p^{(i)})$$

Hence the mathematical expectation of the minimum number of button presses to set the clock (to any time) is

$$\mathrm{E}(n) = \sum_{n=1}^{\infty} nf^{(n)}$$

We can calculate $p^{(n)}$ from the state transition probability matrix M of the deterministic FSA model and the initial state s_0, as

$$p^{(n)} = \mathrm{R}.\mathrm{M}^n s_0\text{, where } \mathrm{R}_i = \begin{cases} 0, s_i \text{ clock stopped} \\ 1, s_i \text{ clock running} \end{cases}$$

We find E(n) and the standard deviation to be 9.6±3.2, that is, more than triple the designer’s “privileged design knowledge” estimate. In the sections following we will indicate improved designs that can be derived directly from the original: such as an “error-blocking” design with E(n) = 6.4±2.3.

On the other hand, if the user assumes the clock is 24 hour, then half of the clock times are unsettable (i.e., the minimum number of button presses for those times is unbounded), so the expected minimum number of buttons is infinite. Random exploration of the system, though worse than well-informed exploration, is better than misguided exploration.

The moral for users, such as ourselves, who get stuck with interactive systems because of incorrect domain knowledge is to flip a coin and act randomly. Had we used a “set breaking” technique such as this, the analysis confirms that we would rapidly have found a successful time setting example. With any valid example, generalizing to setting the time we wanted would have been much easier than persisting at the impasse.

Suppose, further, that we ask the designer what is the expected minimum number of button presses to set the clock, averaged over all clock times. After some thought, he’d say 13.25 presses. The corresponding figure for random exploration is infinite, because for any time that we might want to set, there is a finite (indeed, overwhelming) probability of becoming stuck forever with a different time. Again we see the importance of domain knowledge, and the way it strongly biases the designer to give (dramatically) over-optimistic assessments of his interface.

Incidentally, the designer's figure of 13.25 is much greater than the information-theoretic average minimum with the same button arrangement, $1+\lceil \log_4 720 \rceil$, suggesting that alternative designs should have been considered.

Details of the formulae and calculations used here can be found in (Thimbleby, 1991b).

Error blocking

A problem with the microwave design is that it does not block errors. The user can enter any time (indeed, any number from 00:00 to 99:99). Unacceptable entries such as 12:60 and 15:30 inhibit the expected/desired action of the **clock** button, namely, to stop the colon flashing and start the clock running.

A modified user interface (also tested in HyperCard) with error blocking was remarkably successful. Users could not get into error conditions, and rapidly discovered that the clock was 12 hour. The dramatic success of this design improvement is a corroborative instance of Carroll's minimalist position (1990).

Successful error-free use of any system requires errors to be avoided or blocked *somewhere*, so it is interesting that the "error blocking" in the microwave, such as it is, occurs in a small note in the user manual: it should have been in the system so that the errors simply could not occur! In short, the error handling could have been implemented as computer program (in the microwave firmware) rather than as user program (*qua* user manual to be held in the user's head).

Error supporting

Blocking errors merely stops the user making mistakes, and avoids the potential mess that could have arisen. With the microwave, the user must also discover that once an button press has been blocked, that button is inactive until **reset** is pressed when the colon is flashing.

In what might be called *error supporting*, the actions causing blocks are converted to supportive results. In the clock example, blocking inhibits incrementing the time by one minute from 10:59, since 10:60 is in error. In an error supporting system, on the other hand, incrementing by one minute results in either 10:50 (i.e., incrementing just that digit), or possibly 11:00.

Error supporting raises interesting design tradeoffs. In the above example, domain knowledge tends to suggest that 10:59 be incremented to 11:00. But consider that the button press might have been a mistake. To correct the mistake, it is now necessary to press it 9 more times (obtaining 11:09), then the 10 minutes 5 times (obtaining 11:59), then the 1 hours 11 times (obtaining 10:59). Thus domain knowledge (about arithmetic, such as 9+1 = 10) does not necessarily improve a system: here it makes it worse.

Error supporting has a theoretical advantage. Since "errors" now have non-trivial effects, the user continues to explore the system. In an error blocking system, errors reveal little about the rest of the system. Thus an error supporting system is generally faster to learn, if you don't have the appropriate knowledge (or button pressing skill) to avoid errors. The value of $E(n) = 6.4 \pm 2.3$, the same as the error-blocking system, since the Markov model of the user does not learn.

Undo

Undo is often recommended as — almost — a panacea for improving user interfaces. It is interesting to consider the effect of providing undo in terms of the present approach.

A FSA can be trivially converted to an embedding FSA augmented with a new symbol **undo** and additional states to support the undo functionality. In general $n \leq k$ consecutive presses of **undo** on a k-step undo FSA returns the machine to its state n consecutive nonundo presses earlier. (The special case of a 0-step undo FSA is identical to the original FSA.) If the FSA has N states and S symbols, k-step undo requires $\Omega(NS^k)$ states up to reduction.

If the user knows (from domain knowledge) what **undo** does, then the larger FSA can be explored much faster. If undo is incorrectly implemented (as it often is), or the user does not know its purpose, then providing undo merely has the effect of gratuitously increasing the size of the FSA, and hence combinatorially increasing the time to learn the system.

Timeouts

Timeouts change the state of a machine without explicit input from the user. To understand a machine with timeouts requires domain knowledge about the passage of time, and a view that (for example) only discrete inputs are relevant, and then only on a small part of the state space: otherwise, the increase in complexity of the machine learning problem is unbounded. As we saw with the microwave example, domain knowledge may stop users discovering timeouts; conversely, timeouts may stop the user ever being able to model the machine.

Random exploration of a machine with timeouts is harder. Indeed, if the user attempts to accelerate the state space exploration, some timeout transitions will be missed! In the microwave example, the flashing colon is an example of a timeout: when the colon is flashing, but just happens to be on, it appears equivalent to being stable and on. If the user explores the microwave "too fast," time-dependent features such as colon flashing (indeed, the clock actually running) cannot be observed. The machine therefore appears *vastly* more complex than appropriate domain knowledge might have led one to suppose. In short, the random approach (say, implemented by a robot on a real microwave) is biased against machines with timeouts of any sort.

Information transfer

Consider a long sequence of n button presses out of b: if the button presses are equiprobable, the information transferred into the FSA is $n \log b$. The information out of the machine depends on the observable state probabilities, $-n\Sigma p \log p$, which is maximized when the probabilities are equal. In general, then, it seems that buttons should have equal probabilities of "doing things" (i.e., changing the observable state) to optimize the user's learning from exploring the system. The main problem with the microwave, from this perspective, is that **clock** does too little (it sets the clock in only 7% of the possible states, against the time-setting buttons that each work in 93% of the states). The error blocking and the error supporting machines increase the information transfer rate.

The user interface of a video recorder remote control uses the **transmit** button to do things *too* often (Thimbleby, 1991a), by a factor of around 100. Although the remote control has hidden states, this design defect can be identified using the same procedure as described above.

Hypertext as FSA

Our discussion of automata may have given the false impression that the approach is pragmatically restricted to simple machines, as exemplified by microwave clocks. We now show that the approach applies to "large" systems such as hypertext documents, or to menu-driven systems, etc.

HyperCard is a programmable hypertext system, and we can view each card as a state with buttons (more generally, "go" commands executed from the card) as representing state transitions. A "large" HyperCard stack will have perhaps hundreds of cards, many fewer states than even the microwave's clock of around 10000.

Suppose we have a stack with N cards and b buttons per card that navigate the stack, that is, their scripts contain go commands.

To explore whether each button is correctly scripted means going to each card and pressing every button on it. This requires at least bN button presses and visiting each card at least once. Pressing a button takes us to another card, and we will have to press additional buttons to return to test the outstanding b–1 buttons on the card we have just left. In the worst case, this may mean $O(N^2)$ card visits, though in reality a systematic search could follow a covering of Eulerian paths (requiring at most $2N$ card visits — if you know *a priori* the Eulerian path!), or more simply would make use of HyperCard's back command and hence require $(b+1)N$ card visits. (This is the Chinese Postman Problem.)

For a given HyperCard stack we could calculate such numbers as: given the mean time per card visit, the expected time to visit all cards; or given other assumptions, the expected time to test all buttons. Such times would be huge. User testing of systems will take far too long by empirical means. Hence:

1) Systems must be analyzed systematically, preferably mathematically — and not only to identify bugs.
2) Empirical work *must* use statistical methods (e.g., detecting errors is a random process, and error detection rates might be fitted to a Poisson distribution in order to estimate remaining errors.)

A bonus of systematic analysis, mechanically constructing the FSA from a hypertext, is that this provides an automatic way of checking the stack. Experiments with several stacks showed that one might expect to find 1% errors in go commands, errors such as incorrect syntax and references to non-existent cards. One might also have some cards that are simply unreachable, there being no go links to them. One can also check many non-FSA features: one “read only” stack turned out to have 1% of its text writeable by the user.

If systematic analysis of systems was a routine part of user interface development (as we have argued it should be), then the design of languages for implementing interactive systems might have made analysis a priority. In the case of HyperCard, its programming language HyperTalk could easily have been more carefully designed had tractability been a design goal; instead, it is not only difficult for humans to program in, but very difficult to analyze mechanically, impossible in certain cases (Thimbleby, Cockburn & Jones, 1992).

Back to workstations

Finally, how do we interpret Apple's user interface in the light of our approach? The user views the system as a NDFSA, for example that the operation of dragging a collection of files from one folder to another is a simple state transition. The Apple dialog box forces the user to distinguish between all components of the collection, thereby imposing a combinatorial explosion of states. In a sense there is no wonder that the bug was not diagnosed by user testing.

The intrinsic difficulty of doing a good job

Much work in Human Computer Interaction is motivated by knowledge — or the quest for knowledge — from human psychology, and from anthropology, sociology and other human disciplines. We believe that, necessary as these approaches are, they are not sufficient. There is no point, say, in choosing an easy–to–read font if the underlying system is badly designed. Regardless of representation, there are underlying semantics: and semantics tends to be glossed in the main thrust of contemporary HCI. The received wisdom is to focus on users, participative design, iterative design, and empirical measurement: the bias here is one of specifics about human behavior and performance. Our view is that such approaches tend to *palliate* design problems rather than *solve* them.

Current interfaces certainly need all the palliating they can get, but the more successful this endeavor, the worse the underlying systems can become! The problem is that humans are complex and are easily diverted, and designers end up creating machines like those of the fictional Syrius Cybernetics Corporation (Adams, 1979) whose machines' many superficial flaws completely concealed fundamental design blunders. So long as we concentrate on exclusively human factors we get bogged down in complexity and camouflage (multimedia virtual reality) and thereby miss the fundamental flaws that no amount of disguise can conceal for ever.

This chapter has just scratched the surface of what might be formalized for the purposes of HCI design. This inevitably produces mathematical formulae — does that make HCI design look too complicated? Remember, we've only just started, and future design will only get harder! One might ask, why use formulae when you can build interfaces without them? This question was seriously asked by a negative reviewer of *Formal Methods in Human–Computer Interaction* (Harrison & Thimbleby, 1990).

Consider almost any other design discipline, say, architecture. Part of a building's design may include a beam. The material of the beam, its shape, supports and so forth must be carefully designed. There are cases of buildings collapsing because the designers failed to take account of various design variables, such as the mass of the beam itself. Such variables have to be manipulated by skilled engineers in the appropriate — and difficult — formulae. There are many cases of buildings collapsing because aesthetic considerations overrode mathematical and practical engineering considerations (Petroski, 1985). Buildings that collapse may be unfortunate combinations of circumstances, the result of an incautious extension too far beyond current safe practice, or simple, professional incompetence.

When you build a shed in your backyard it is almost inevitably over-engineered; certainly when you build a $^1/_{72}$ scale model, it is far stronger than necessary! When you build a skyscraper, every effort has to be spent on mathematical modelling: not only are the risks greater, but the error margins are smaller.

In the design of interactive computer systems there is no feel for scale (Thimbleby, 1990). A child can write a program that runs on a large computer. A teenager might write an arcade game that looks like a missile control program. The scale of a program, and the professional skill of its designer, are both hidden.[4] Given human nature it is not surprising that the temptation to hire the cheapest or fastest programmer is given precedence to hiring one who is competent, particularly since system users probably won't find out which — certainly not until the so-called warranty has run out (usually as soon as the box is opened). The problem is so deeply ingrained that we don't even know what a competent programmer would be like (despite Dijkstra, 1972).

Our concern could progress from interactive systems to star wars (Parnas, 1985). Yet designers of interactive computer systems have fewer excuses than architects. An architect does not plan the geology nor the physical properties of building materials. In comparison, the hardware of computer systems is unbelievably reliable; indeed all the design errors we have been concerned with in this chapter have been concerned with software.

We rhetorically asked why user interface design was so complex. We end this section by asking why, when in comparison with other designers HCI designers have a better chance of doing a good job, they avoid formal reasoning? — like architects who worry about paint, hoping the joists will look after themselves. Why has practically nothing semantic been done in HCI since Moore's 1956 paper or Gaines' 1975 paper? Why aren't standard techniques — such as "design for testability" (e.g., Weste & Eshraghian, 1985) — adopted in HCI?

Look and feel

There is considerable ferment in the HCI community about "look and feel" — should it be copyright? Where are the real ideas and innovations? Who should benefit: users or

[4] By 'scale' we are not referring to the size of the program in bytes: the combinatorial possibilities are *much* greater, leading to severe complexity under-estimates even, perhaps especially, when the designer knows the size.

designers? And so on. When professional programmers happily publish real systems' Pascal code (Bond, 1988) typified by[5]

```
VAR exitLoop: Boolean;
{ code omitted }
exitLoop := FALSE;
{ code omitted }
REPEAT
  IF a THEN exitLoop := TRUE
  ELSE b
UNTIL exitLoop = TRUE;
```

— which is just a spread-out, inefficient and error-prone circumlocution for `while not a do b`, which does not need the flag (indeed, in some code `exitLoop` is *both* a local and global variable!) — it is not difficult to imagine why there is such pressure to keep the HCI argument centered in "look and feel," away from the significant issues of computer science like correctness and efficiency.

Our view is that the look and feel is literally superficial, and the real issues in HCI are computational. So far, as is obvious from the discussion, the computational issues are difficult to even start to debate! But it is rather like supposing the early car manufacturers attacked each other for copying "look and feel" widgets like steering wheels (some cars had tillers, some had handlebars), when the real level at which cars competed was in performance: what they could *do* rather than *how* they interacted with or appeared to the user. Successful cars were more reliable, faster, more fuel efficient, and so on. Such functional concepts — like reliability — are still far from the center of HCI, and will remain so until designers put semantics first.

Why is it that software manufacturers compete with each other over look and feel, but give the user (at best) a 90 day warranty on the discs, *no* warranty on the correctness of the system underlying the user interface? Conventional HCI methods will add layers to *conceal* fundamental problems, and permit the implementations to be yet more unreliable. A random exploration of a system (whether by a robot or a mathematician!) ignores cosmetics and gets to the basis of user interaction: it does not make user interface design any easier, but it will encourage designers to seek improvements where it really matters in the long run.

Conclusions

When we see that to test a simple FSA such as *just* the clock setting part of a microwave oven takes 40000 button presses (almost a month's work for a human pressing buttons twice a minute, 12 hours per day, and no mistakes!), one might despair of ever being able to test such things! Likewise, to learn such a system without prior domain knowledge will take — waste? — a similar sort of time. On the other hand, a bamboo flute (with 8 holes against 6 keys) requires exploration of only 256 combinations of button presses *at most*, yet many people spend a lifetime improving their skill and presumably their mental model of the trivial FSA it represents.

User interface design is *far* more complex than we might wish, because interactive systems are very complex. It is time that we stopped palliating deep problems with superficial features. This conventional approach is bound to backfire as the implementations of systems become more complex and more concealed. If a system (e.g., a word processor) is made nicer (in some human sense) and the users do more work with it, that merely means that the disasters will be worse, and the expected information loss

[5] This is not the place to give large examples.

will be greater. Regret will be compounded: it *seemed* better, but now is far worse. Thus systems have to be made nicer in a computational sense: they have to be easier to model.

Acknowledgements

The work reported in this chapter was supported by a grant from the Canadian National Science and Engineering Research Council during a visit of one of the authors to Calgary University, Alberta. Our thanks go to Saul Greenberg for the loan of the microwave oven that began it all. Brian Gaines and Saul Greenberg both made very helpful comments on the text for which we are grateful.

References

D. Adams (1979) *The Hitchhiker's Guide to the Galaxy*, Pan Books.

Apple, Inc. (1987) *Human Interface Guidelines: The Apple Desktop Interface*, Addison-Wesley.

G. Bond (1988) *XCMD's for HyperCard*, Portland: Management Information Source, Inc.

J. M. Carroll (1990) *The Nurnberg Funnel*, Massachusetts: MIT Press.

J. H. Conway (1971) *Regular Algebra and Finite Machines*, London, England: Chapman and Hall.

P.–J. Courtois (1985) On Time and Space Decomposition of Complex Structures, *Communications of the ACM*, **28**(6), pp.590–603.

E. W. Dijkstra (1972) The Humble Programmer, *Communications of the ACM*, **15**(10), pp.859–866.

B. R. Gaines (1971) Memory Minimization in Control with Stochastic Automata, *Electronics Letters*, **7**, pp.710–711.

B. R. Gaines (1975) Behaviour/Structure Transformations Under Uncertainty, *International Journal of Man–Machine Studies*, **8**, pp.337–365.

B. R. Gaines (1977) System Identification, Approximation and Complexity, *International Journal of General Systems*, **3**, pp.145–174.

M. D. Harrison & H. W. Thimbleby (1990) *Formal Methods in Human–Computer Interaction*, Cambridge: Cambridge University Press.

B. P. Miller, L. Fredriksen & B. So (1990), An Empirical Study of the Reliability of Unix Utilities, *Communications of the ACM*, **33**(12), pp.32–44.

M. Minsky (1967) *Computation: Finite and Infinite Machines*, Englewood Cliffs, NJ: Prentice-Hall.

E. P. Moore (1956), Gedanken Experiments on Sequential Machines, in C. E. Shannon & J. McCarthy, eds., *Automata Studies*, Princeton: Princeton University Press, pp.129–153.

D. L. Parnas (1985) Software Aspects of Strategic Defense Systems, *Communications of the ACM*, **28**(12), pp.1326–1335.

Panasonic (undated) *Operating Instructions Microwave/Convection Oven Model NN-9807/NN-9507*. Matsushita Electric of Canada Ltd., Ontario.

Panasonic (1991) *Microwave Cookery Book Models NN5800/5850/5400/5450*, Matsushita Electric of Canada Ltd., Ontario.

H. Petroski (1985) *To Engineer is Human: The Role of Failure in Successful Design*, MacMillan.

C. Plaisant & B. Shneiderman (1991) Scheduling ON–OFF Home Control Devices, in *Reaching Through Technology*, CHI'91 Conference Proceedings, pp.459–460.

H. W. Thimbleby (1990) *User Interface Design*, Reading: Addison-Wesley.

H. W. Thimbleby (1991a) Can Anyone Work the Video? *New Scientist*, **129**(1757), pp.48–51.

H. W. Thimbleby (1991b) Formal Methods with No Psychology, Proceedings Institution of Electrical Engineers Colloquium, *Theory in Human-Computer Interaction*, 1991/192.

H. W. Thimbleby, A. Cockburn & S. R. Jones (1992) HyperCard: An Object Oriented Disappointment, in *Building Interactive Systems: Architectures and Tools*, P. D. Gray & R. Took, editors, Springer-Verlag.

N. Weste & K. Eshraghian (1985) *Principles of CMOS VLSI Design*, Addison-Wesley.

Appendices

A1. A specification of the microwave

The microwave clock-setting can be formally described as a FSA with enough states to cover the observable range. A minimal FSA for it therefore has 10721 states and 6 symbols, abstracting out the temporal nature of the colon flashing and the time running. (The FSA model is time invariant.) If the microwave however had hidden states (e.g., that the clock button must have been pressed at least twice before the clock can be set running) then the number of states would be greater than the number of observable states.

Rather than FSA tuples, variable-free production rules are a more convenient specification of the microwave clock (and are provided below for completeness); there is, however, no reason to suppose that they are an optimal representation for any system for capturing issues in usability. On the contrary, the lack of variables would be a serious problem for specifying systems with hidden states. Even for the microwave clock, a notation including arithmetic would be more conducive than regular expressions.

Rules have the form:

button–press conditions → *change*

The conditions are regular expressions, that is, representations of FSAs. Unchanged state is not indicated (nor the associated beeps that the microwave sometimes makes). For clarity in what follows, we name buttons more clearly than on the device itself; represent leading zeroes as 0 (not blank); and as notational device, indicate unaffected clock numerals by ϕ.

Switch-on → stable 00:00 clock-off

Clock^2 stable clock-running → flashing 00:00 clock-off

Clock stable clock-off → flashing

Clock flashing (0[1–9]:[0–5][0–9])|(1[12]:[0–5][0–9]) → stable clock-running

Reset flashing → 00:00

10hour flashing 0ϕ:ϕϕ → 1ϕ:ϕϕ

(8 similar rules omitted.)

10hour flashing 9ϕ:ϕϕ → 0ϕ:ϕϕ

1hour flashing ϕ0:ϕϕ → ϕ1:ϕϕ

(8 rules omitted.)

1hour flashing ϕ9:ϕϕ → ϕ0:ϕϕ

(20 rules for 10minute and 1minute omitted.)

Much of the complexity is restricted to one rule: states are not equiprobable. For example, there are many ways of getting in the 00:00 state (**reset** gets there from any state with a flashing colon), whereas the state, say, 34:67 is only reached from 4 other states.

The restriction on Clock flashing can be distributed more evenly amongst the rules that affect its firing, this results in an error blocking or an error supporting system, and saves around ten rules depending on design decisions. We give example error supporting rules:

Switch-on → flashing 01:00 clock-off
Clock^2 stable → flashing 01:00 clock-off
Clock flashing → stable clock-running
Reset flashing → 01:00
10hour flashing 0[1–2]:ϕϕ → 1ϕ:ϕϕ
10hour flashing 0[3–9]:ϕϕ → 10:ϕϕ
10hour flashing 10:ϕϕ → 01:ϕϕ
10hour flashing 1[1–9]:ϕϕ → 0ϕ:ϕϕ
1hour flashing ϕ0:ϕϕ → ϕ1:ϕϕ
1hour flashing ϕ1:ϕϕ → ϕ2:ϕϕ
(7 rules omitted.)
1hour flashing 09:ϕϕ → 01:ϕϕ
1hour flashing 19:ϕϕ → 10:ϕϕ
(15 rules for 10minute and 1minute omitted.)

A2. Software simulation

The HyperCard stack mentioned in this chapter was developed using HyperCard version B1-2.1. In the interests of facilitating empirical duplicating our investigations, the code is as follows:

```
Script of Microwave Simulation as on Wednesday, December 4, 1991

» Script of Card
on opencard
  global off, flash, last_click_time
  set userlevel to 5
  put true into off
  put false into flash
  put 0 into last_click_time
  hide card picture -- make it look unplugged in
  set the name of card button ID 7 to "Off"
  notime
end opencard

on notime
  global clock_running
  put false into clock_running
  put "     " into field "time"
end notime

on idle
  global flash, off, start_time, clock_running
  if off then exit idle
  if flash then
    get char 3 of field "time"
    if it = ":" then -- this may flash too fast!
      put space into char 3 of field "time"
    else
      put ":" into char 3 of field "time"
    end if
  else
    if clock_running then
      get the secs-start_time
      convert it to short time -- now in the form "12.35 AM"
      get word 1 of it
      if number of chars in it < 5 then put space before it
      put it into field "time"
    end if
  end if
end idle
```

```
function oktime t  -- check a time is valid
  get char 1 to 2 of t -- hours
  if it < 1 or 12 < it then return false
  get char 4 to 5 of t -- minutes
  if it < 0 or 59 < it then return false
  return true
end oktime

on digit d -- increment a digit in the display
  global flash, off
  if off or not flash then exit digit

  get char d of field "time"+1
  if it > 9 then put 0 into it
  put field "time" into t
  put it into char d of t

  get true
  repeat with i = 1 to 5 -- fix leading zeroes
    if char i of t is in " 0" and it then
      put space into char i of t
    else
     if i ≠ 3 then
       get false
       if char i of t = space then put 0 into char i of t
     end if
    end if
  end repeat

  put t into field "time"
end digit

-- These two handlers are common operations for all the buttons;
-- but check that the mouse target IS a button.

on mousestilldown -- implement autorepeat on the buttons
  global off
  if not off and word 2 of name of the target = "button" then
    set the hilite of the target to true -- if not already set
    if the ticks mod 60 = 0 then send mousedown to target
  end if
end mousestilldown

on mouseup
  if word 2 of name of the target = "button" then set the hilite of the target to false
end mouseup
```

» Script of button “Off”

```
on mouseup
  global off, clock_running
  if not off then opencard -- off
  else -- on
    put false into clock_running
    put false into off
    show card picture
    set the name of me to "On"
  end if
  pass mouseup
end mouseup

on mousestilldown
  -- don't let message pass to card script
end mousestilldown
```

» Script of button “Clock”

```
on mouseup
  global flash, off, start_time, last_click_time, clock_running
  if off then exit mouseup
  if the ticks – last_click_time ≤ 120 then -- double click clears clock
    put true into flash
    notime
  else
    if flash then
      put not oktime(field "time") into flash
      put not flash into clock_running
      if clock_running then
        put ":" into char 3 of field "time" -- colon might have been off
        get field "time"
        convert it to seconds
        put the secs - it into start_time
      end if
    else
      if not clock_running then put true into flash
    end if
  end if
  put the ticks into last_click_time
  pass mouseup
end mouseup
```

» Script of button “Reset”

```
on mouseup
  global flash, off, mouseup_time
  put 0 into mouseup_time
  if not off and flash then notime
  pass mouseup
end mouseup
```

» Script of button “10”

```
on mousedown
  digit 1
end mousedown
```

» Script of button “1”

```
on mousedown
  digit 2
end mousedown
```

» Script of button “10”

```
on mousedown
  digit 4
end mousedown
```

» Script of button “1”

```
on mousedown
  digit 5
end mousedown
```

» Script of button “Socket”

```
-- some fun if the user puts their fingers in the electrical socket
on mouseUp
  global off
  if off then -- the socket is exposed to fingers …
    play "explosion"
    play "ARRGH!"
    visual effect dissolve fast to black
    visual effect dissolve fast to white
    visual effect dissolve fast to black
    go this card
  else
    opencard -- switch Microwave off
  end if
  pass mouseup
end mouseup
```