Specification-led design

for interface simulation, collecting use-data, interactive help, writing manuals, analysis, comparing alternative designs, *etc*.

Wednesday, September 30, 1998

Harold Thimbleby
Middlesex University
LONDON
N11 2NQ

URL: http://www.cs.mdx.ac.uk/harold

Abstract

This paper shows how to combine a substantial part of the product development cycle of interactive devices into a single, co-ordinated approach. Much can be derived automatically from a suitable specification of the interactive device, and it can be derived automatically. Normal product development has a device specified and built, then has its manuals written, then it is used and tested. At this late stage design problems may be identified, but it is now too late: usability studies become academic in so far as the particular product is concerned, since it is already effectively in production. It would be better if the testing and manual writing could rapidly be obtained from the initial specification, before any investment has been made in fabrication. This paper offers a design approach that achieves this, and it shows how the various views of the design can be used help improve each other — for instance, the automatically generated user manual can be fed back to suggest improvements in the design.

A microwave cooker is used as a real example. However, this paper provides full and unabridged details of everything it discusses by using *Mathematica* as a rapid prototyping environment. Any similar device can be analysed in the same way, directly from the paper.

Introduction

This paper shows that the definition of a device, its simulation, its usability analysis, and its user manuals in any language (and interactive help, if required), can all be worked on directly and efficiently in a suitable design environment. If part of the design process suggests improvements, say that the user manual has an obscurity, then it can be changed directly by modifying the specification; and the new specification will update all other parts of the product, the analysis, the simulation, and so on. Importantly, the approach only has one definition of the device; thus changes — as occur during iterative design and product revision — immediately and automatically affect all parts of the development process: the analysis, the simulation, the help (and even the hardware). Specification-led design is so efficient that it is effectively concurrent engineering.

The importance of the approach is that many components of a product are derived efficiently and automatically, almost at once. In normal design methods, there is a sequential (and costly) progression from specification, through fabrication, to manual writing, and finally usage. Only at later stages, then, will many usability problems be identified — but by then, the product is already fabricated, and many of the usability insights would be very hard to take back to the specification, even it was still available.

For this paper, to illustrate our approach we have used the symbolic mathematical system, *Mathematica* (Wolfram, 1996). *Mathematica* is sufficiently powerful that everything described in this paper is completely and fully specified here. Many popular system development environments, especially those aimed at providing visual realism, would have been inappropriate because they never explicitly 'know' what the device specification is, and none of the automatic benefits discussed here could have been be obtained. This paper was developed and printed entirely within *Mathematica*: all the examples are genuine, and have not been re-keyed or fudged in any way. Stylistically, this does have the disadvantage for this paper that explaining the method is interleaved with potentially distracting explanation of the *Mathematica* code. If desired, the approach could be packaged (e.g., in Java) in such a way that none of the technical details would be visible to a designer. However, we felt that for the purposes of this paper, being able to see that the method works, and giving completely open details was more important. Not only does this paper claim that specification-led design is a useful approach, but it provides *complete* details for anyone to copy the approach.

As a running example, we shall use the definition of a microwave cooker, as given in Jonathan Sharp's PhD thesis (Sharp, 1998). As a deliberate decision, we used Sharp's *exact* definition, to try and emphasise the generality of the approach (we didn't choose this definition to fit the needs of the paper).

■ Preliminaries

Since this paper **is** the *Mathematica* definition of everything the paper talks about, it has to start with some *Mathematica* preliminaries. This section can be skipped, as they say, on first reading — though it also provides some examples of *Mathematica* being used, to explain the formatting conventions this paper uses. A stylistic consequence of this paper also being a *Mathematica* document is the rather frequent use of backward references in the text to earlier definitions — they have to be earlier, else they wouldn't work where they are needed!

We start 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 a basic utility routine.

```
<< DiscreteMath`Combinatorica`;

vector_ ' element_ := Position[vector, element][[1, 1]]
```

The function ' (pronounced 'such that,' defined above) gives the numerical index into a vector such that the element would be selected. (It uses the built-in function `Position`, which produces a general result, which in turn requires the `1,1` subscript.) Here is how it is used:

```
exampleVector = {firstElement, secondElement, thirdElement};

exampleVector ' secondElement
```

2

Here we see an example of *Mathematica* output: `secondElement` is to be found at position 2 in the example vector. And, working the other way round, the 2 can be used to select the second element of the vector:

```
exampleVector⟦2⟧
```

secondElement

In all cases in this paper, *Mathematica* code precedes the output from running the code. The cases where no output is shown are either straight-forward definitions, or are stated in the text as not being run (for example, to save space, we did not run the *Mathematica* code to generate the *entire* user manual).

■ Device definition

Mathematica shows how easy reusable development is to do. By making minor changes to the definitions here, other devices can be developed in the same way.

Here is Jonathan Sharp's definition of his microwave cooker. Because we decided to use *exactly* his definition (for reasons given above), the function ' (defined above) is used frequently to convert between names and numbers; had Sharp defined his device directly in terms of state numbers this would not have been necessary. (Instead, we might have defined each button and state as a numerical constant; but the approach we have used makes the device definition easier to read and less error-prone.)

```
device = {
  {clock, clock, clock, clock, clock, clock},
  {quickDefrost, quickDefrost, quickDefrost,
   quickDefrost, quickDefrost, quickDefrost},
  {timer1, timer1, timer2, timer1, timer2, timer1},
  {clock, clock, clock, clock, clock, clock},
  {clock, quickDefrost, power1, power2, power1, power2}
};

buttonNames = {clock, quickDefrost, time, clear, power};
stateNames =
 {clock, quickDefrost, timer1, timer2, power1, power2};

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

The five parts of the device specification, here represented by five variables (`buttonNames`, `numberOfButtons`, *etc.*), can be encapsulated into a single structure, and for actual development work this would have been preferable, rather than proliferating five variables per design. *Mathematica* provides various ways to do this (packages, object-oriented programming, *etc.*), but for such a brief paper as this, to do so would introduce unnecessary technical detail.

Sharp didn't write his specification in *Mathematica*! *Mathematica* can, however, print the specification above quite closely to the style that Sharp used; in fact, *Mathematica* provides an extensible user interface to make the entry of tabular data as easy as using a spreadsheet.

We define a function `neatTable` to make a reasonably neat tabular presentation of any device. It is probably clear from the intricacy of this code that almost any typographical details can be accommodated.

```
neatTable[title_, device_, stateNames_, buttonNames_] :=
 With[{heading = StyleBox[#, FontFamily Æ "Helvetica",
      FontSize Æ 10, FontWeight Æ "Bold"]&,
   subHeading = StyleBox[#, FontWeight Æ "Bold"]&},
  StyleBox[GridBox[{{FrameBox û GridBox[
         {{GridBox[Transpose û {Join[{heading["Buttons"], ""},
               subHeading /û buttonNames]},
            ColumnAlignments Æ Right],
           GridBox[ {{heading["— States —"]},
             {GridBox[Join[{subHeading /û stateNames},
                device], ColumnLines Æ True,
              RowLines Æ {True, False}]}}]]}},
        ColumnLines Æ True]},
      {heading[title]}}],
    FontFamily Æ "Palatino",
    FontSize Æ 9, FontWeight Æ "Plain"] // DisplayForm];

neatTable[
 "Sharp's Microwave cooker", device, stateNames, buttonNames]
```

Buttons	— States —					
	clock	**quickDefrost**	**timer1**	**timer2**	**power1**	**power2**
clock	clock	clock	clock	clock	clock	clock
quickDefrost	quickDefrost	quickDefrost	quickDefrost	quickDefrost	quickDefrost	quickDefrost
time	timer1	timer1	timer2	timer1	timer2	timer1
clear	clock	clock	clock	clock	clock	clock
power	clock	quickDefrost	power1	power2	power1	power2

Sharp' s Microwave cooker

The table is read as follows: choose the column according to the current state of the device, then read off the next state of the device from the row corresponding to the button pressed.

If we hadn't wanted all the typographical details just so, *Mathematica* could have printed the specification in a basic form, just with `TableForm[device]`.

■ Simulating the user interface

To simulate the device, we use a global variable to keep track of the changing state of the device as buttons are pressed. We will start the device in state 1, which happens to be `clock`. Arguably, a device definition should specify its initial state — the state a device is in as soon as it is used: for many devices, this state will be its being off.

```
state = stateNames⟦1⟧
```

clock

The definitions given in this section merely show the name of the current state in the display. It is possible to display anything, not just plain text, but to do so would take us beyond the scope of this paper.

When a button on the simulation is pressed, *Mathematica* will arrange for the function `press` to be called, with the button as a parameter.

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

After collecting any useful statistics, this function uses the `device` specification to determine the next state. The next few lines of the function locate the device's display cell in the current *Mathematica* notebook (*Mathematica* can have several notebooks — that is, windows — running together, which is why the variable `nb` is required); the text displayed in that cell is selected and replaced with the name of the new state.

By defining `collectStatistics`, we make `press` collect empirical statistics as the simulation is used. For simplicity, we will just collect state transition counts:

```
statistics = Table[0, {numberOfButtons}, {numberOfStates}];

collectStatistics[theButton_, state_] :=
 ++statistics[[buttonNames ' theButton, stateNames ' state]]
```

The following code is the definition of a row of buttons to control the device.

```
   Cell[BoxData[
    RowBox[{
      ButtonBox["Clock",
        ButtonFunction:>press[clock],
        ButtonEvaluator->Automatic],
      ButtonBox["Quick defrost",
        ButtonFunction:>press[quickDefrost],
        ButtonEvaluator->Automatic],
      ButtonBox["Time",
        ButtonFunction:>press[time],
        ButtonEvaluator->Automatic],
      ButtonBox["Clear",
        ButtonFunction:>press[clear],
        ButtonEvaluator->Automatic],
      ButtonBox["Power",
        ButtonFunction:>press[power],
        ButtonEvaluator->Automatic]}]],
  Active->True]
```

To use the buttons, *Mathematica* would change the display mode of the definition, and show a row (by default) of actual buttons:

Clock | Quick defrost | Time | Clear | Power

The device's simulated display is a simple *Mathematica* 'cell' (shown below) with an appropriate name so that the `press` function can locate it. In its simplest form it could be just `Cell["", CellTags Æ "display"]`.

If desired, *Mathematica* allows cells and buttons to contain further 'typographical' details, such as their font, size and colour. For example, the device's display can easily be made to look more like a typical LED display of green text on a black background, by providing options (such as, `FontFamily Æ "Courier", FontColor Æ RGBColor[0, 1, 0], Background Æ GrayLevel[0]`, and its correct dimensions) in the definition of the cell:

In a running *Mathematica* session, pressing the buttons makes this display work, as well as collect data on the users' behaviour with the simulation.

Mathematica can itself generate button definitions from *any* device specification, and one can extend the definition to include explicit sizes, positions and so forth. (We will give an example below.) Thus, the user interface itself can be defined by the *same* device specification. This is very important to make the analysis — both mathematical and empirical — use consistent specifications; they can be edited easily and only in *one* place.

■ Analysis and graph drawing

For illustrative purposes, we now do a Markov analysis of the device, which is a good way of estimating how a user, making random errors, would perform using the device. It should be noted that this approach is a 'keystroke level model' but which allows for errors. In particular, we will be able to draw a graph of a user's task performance against how accurately (how error-free) or how well they know how to do the task perfectly.

A working paper, available from the author, is available to describe the particular benefits, and many further details, of the approach (Cairns, Jones & Thimbleby, 1998). Published papers further explaining the motivation for such analyses are Thimbleby & Witten (1993) and Thimbleby (1994).

We will analyse the user task of getting from state `power1` to state `power2`. To consider a particular task, we do need to know the appropriate state names. Alternatively, it is possible to analyse all pairs of states (hence, all tasks the device supports) and obtain statistics, which would typically be weighted by the relevance or importance of the tasks to users. However, for the purposes of this paper, analysing just one task is sufficient.

```
start = stateNames ' power1;
goal = stateNames ' power2;
```

We now convert Sharp's definition into a stochastic matrix:

```
randomUser = Table[0, {numberOfStates}, {numberOfStates}];

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

Here is the matrix displayed in traditional mathematical notation:

```
randomUser // TraditionalForm
```

$$\begin{pmatrix} \frac{3}{5} & \frac{1}{5} & \frac{1}{5} & 0 & 0 & 0 \\ \frac{2}{5} & \frac{2}{5} & \frac{1}{5} & 0 & 0 & 0 \\ \frac{2}{5} & \frac{1}{5} & 0 & \frac{1}{5} & \frac{1}{5} & 0 \\ \frac{2}{5} & \frac{1}{5} & \frac{1}{5} & 0 & 0 & \frac{1}{5} \\ \frac{2}{5} & \frac{1}{5} & 0 & \frac{1}{5} & \frac{1}{5} & 0 \\ \frac{2}{5} & \frac{1}{5} & \frac{1}{5} & 0 & 0 & \frac{1}{5} \end{pmatrix}$$

Each row gives the probability that the user will change the state; thus, if the device is in state 1, the user will change it to state 2 with probability 1/5 (i.e., first row, second column).

This matrix will be used for the analysis. The assumption is that each button on the device is pressed with equal probability. (There are five buttons, so all the probabilities are so-many fifths.) The user interface simulation can give empirically-based probabilities, and we will analyse them below.

Here we give the definition of the mean first passage time in its most direct form (our associated paper gives a full derivation of the relevant formula). The mean first passage time represents a user's difficulty with performing a task.

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

One = Table[1, {numberOfStates}];
Id = IdentityMatrix[numberOfStates];

meanFirstPassage[matrix_, start_, goal_] :=
 (Inverse[Id - ZeroRowCol[matrix, goal]] . One)[[start]];
```

Here is how the function can be used:

```
meanFirstPassage[randomUser,  start, goal]
```

```
120
```

Thus, the expected time to perform the task, to get from the start state (`power1`) to the goal state (`power2`), is 120 button presses. Of course, the Markov model doesn't "know" how to use the microwave, which is why the number seems so high. But the designers of devices should know how to use them! We now create a designer's matrix, which represents optimal use for any task, based on the optimal route from the start to the goal states. (The random user matrix is converted to a `Graph` type to

find shortest paths; it can be done conveniently — and correctly! — from the `randomUser` matrix, since exactly its non-zero elements are device transitions.)

```
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}];
```

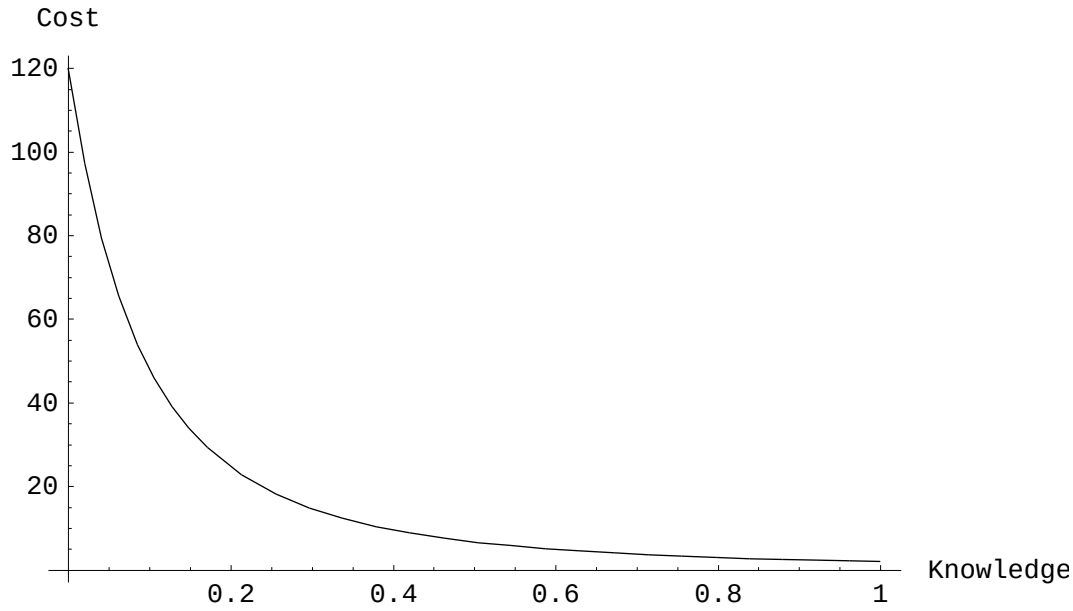
We should check that this designer knows how to do the task!

```
meanFirstPassage[designer, start, goal]
```

```
2
```

Evidently, the more knowledge the easier the device is to use. A graph of difficulty of use against knowledge can be plotted:

```
Plot[
 meanFirstPassage[x designer + (1 - x) randomUser, start, goal],
 {x, 0, 1}, AxesLabel → {"Knowledge", "Cost"}];
```



This shows that as a user learns more about the device (the larger x), until they know as much about it as the designer (x=1), their performance improves. In particular, if the user doesn't know much (where the graph is steep), then even a little help can have a dramatic improvement on their performance. We don't have enough space to do this sort of graph justice, except to point out different device designs (which can easily be explored) have different shaped curves, and hence this approach gives useful insight into design trade-offs.

Many other sorts of analysis are possible. See Thimbleby (1994), a general approach, and Thimbleby (1997), the analysis of a particular device, for further examples. Below, we shall show that it is possible to generate user manuals from specifications: the structure of manuals can be analysed (without anyone ever having to *see* them) — for example, to identify the hardest (e.g., most lengthy) parts of them, and

then to redesign the device so that awkward parts are simplified. Thimbleby & Addison (1994) show how to use flow analysis, arguing that user manual design should follow program design best practice.

■ Looking at empirical statistics of use

The statistics of use collected could be used directly in link analysis and with other conventional design techniques (Stanton, 1998), but we shall continue with the Markov analysis. We could use the function `neatTable`, defined above, to print out the statistics in a neat form.

The actual statistics data used to calculate the information in this section is shown below. (In fact, the numbers here were originally printed by asking *Mathematica* for the value of `statistics` during a session.) The *Mathematica* code below can be run to initialised the variable `statistics` (e.g., during a live demonstration of this paper).

```
statistics = {{2, 4, 1, 1, 0, 0}, {5, 2, 1, 2, 1, 0},
  {8, 3, 6, 0, 0, 1}, {4, 2, 2, 2, 1, 0}, {6, 1, 2, 1, 0, 0}};

neatTable["Button presses in each state", statistics,
 stateNames, buttonNames]
```

Buttons	— States —					
	clock	quickDefrost	timer1	timer2	power1	power2
clock	2	4	1	1	0	0
quickDefrost	5	2	1	2	1	0
time	8	3	6	0	0	1
clear	4	2	2	2	1	0
power	6	1	2	1	0	0

Button presses in each state

Which is the most popular button?

```
With[{b = Map[Apply[Plus, #]&, statistics]},
 buttonNames[[Position[b, Max[b]][[1, 1]]]]
```

time

We will explore alternative designs below, and in particular we shall look at the significance of the `time` button to usability.

We can ask how well the users of the simulation performed the task. The `statistics` matrix counts button presses in each state; we now convert it to a transition matrix; each row of it has to be divided by the total number of transitions out of the corresponding state, to convert the matrix to a stochastic matrix (each row adds to a probability of 1):

```
statsMatrix = Table[0, {numberOfStates}, {numberOfStates}];

Do[
 statsMatrix[[i, stateNames ' device[[b, i]]]] += statistics[[b, i]],
  {b, numberOfButtons}, {i, numberOfStates}];

N[statsMatrix =
   statsMatrix / Map[Apply[Plus, #]&, statsMatrix], 2] //
 TraditionalForm
```

$$\begin{pmatrix} 0.48 & 0.2 & 0.32 & 0 & 0 & 0 \\ 0.5 & 0.25 & 0.25 & 0 & 0 & 0 \\ 0.25 & 0.083 & 0 & 0.5 & 0.17 & 0 \\ 0.5 & 0.33 & 0 & 0 & 0 & 0.17 \\ 0.5 & 0.5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1. & 0 & 0 & 0 \end{pmatrix}$$

```
N[meanFirstPassage[statsMatrix, start, goal], 2]
```

```
61.
```

Since this is better than ignorance (120 steps), but worse than the designer's optimal, it is likely that this user (or collection of users) sometimes did the required task, or almost did it, but whatever they did was not as random as knowing nothing!

We can determine how thoroughly the user interface simulation has been tested; perhaps some transitions have not been tried out by any user so far? We could use *Mathematica* to summarise the as-yet untested transitions. It may be that by getting users to try these transitions out that we discover some obscure behaviour in the device. Perhaps some of the transitions the device supports are counter-intuitive? The following simplistic code tells us what buttons users have not yet been tried (it doesn't try to produce good English!).

```
Do[If[statistics[[b, s]] == 0, Print["Nobody tried to press ",
   buttonNames[[b]], " when in state ", stateNames[[s]]]],
 {b, numberOfButtons}, {s, numberOfStates}]
```

```
Nobody tried to press clock when in state power1

Nobody tried to press clock when in state power2

Nobody tried to press quickDefrost when in state power2

Nobody tried to press time when in state timer2

Nobody tried to press time when in state power1

Nobody tried to press clear when in state power2

Nobody tried to press power when in state power1

Nobody tried to press power when in state power2
```

What transitions did the users try, but which the device isn't designed to support?

```
Do[If[statistics⟦b, s⟧ π 0 && device⟦b, s⟧ === stateNames⟦s⟧,
  Print[buttonNames⟦b⟧, " was pressed in state ",
   stateNames⟦s⟧, " but did nothing"]],
 {b, numberOfButtons}, {s, numberOfStates}]
```

```
clock was pressed in state clock but did nothing

quickDefrost was pressed in state quickDefrost
  but did nothing

clear was pressed in state clock but did nothing

power was pressed in state clock but did nothing

power was pressed in state quickDefrost but did nothing
```

More sophisticated analysis would likely use a log of the users' button presses, whereas the statistics collected in the function `press` only counted state changes — this throws away the information about which button is pressed, and it also loses information relating to tasks that take more than one button press.

Exploring alternative designs

Specification-led design is ideal to explore trade-offs for alternative designs. Obvious alternatives for Jonathan Sharp's device would be to explore designs that have one button per state (so buttons change the state of the device predictably), or to have a single button that cycles through all states. Both of these alternative designs are simple, but are only appropriate for a device with a small number of states. This section of the paper shows how we can explore some alternative design ideas that would also be appropriate for devices with much larger number of states. For clarity, we will not introduce new device specifications, just different ways of interacting with the original device.

The mean first passage time says how many button presses a user takes. From the graph, it is clear that an ignorant user, one behaving quite randomly, is very inefficient, taking 120 button presses — to do a task that a knowledgeable user can do in just 2 presses. Can we modify the design so that 'ignorant' users are more efficient? Much of their inefficiency comes about because they press buttons that do nothing. Let us modify the design so that users are discouraged from pressing pointless buttons. We could imagine that each button can be lit up, perhaps so that its name is only visible when its light is on. (If the device was like a video recorder, it would most often be used in the dark anyway, so lights on buttons would have a dramatic effect on users' behaviour.)

To analyse this new design, we construct a new matrix, `litButton`, that represents the (random) behaviour of users who only press buttons that do something. The matrix can be calculated from the random pressing matrix (used above), by zeroing the diagonal (presses that do not change state) and renormalising:

```
litButton = Table[If[i ≠ j, randomUser⟦i, j⟧, 0],
  {i, numberOfStates}, {j, numberOfStates}];
litButton =
 Table[litButton⟦i⟧ / Plus ûû litButton⟦i⟧, {i, numberOfStates}];

N û meanFirstPassage[litButton,  start, goal]
```

70.8

This is an improvement on 120, which suggests we should do some empirical experiments with users. To do so, we can revise the *Mathematica* simulation and arrange for buttons to change colour depending on whether they actually do anything in the current state.

We define `newInterface` to be an expression that *Mathematica* can render as a row of coloured buttons, but for the time being we don't choose any particular colours. Instead, `RGBPlaces` records the 'slots' where the colour specifications are needed, so the colours can be updated every time a button is pressed.

```
newInterface = Cell[BoxData[RowBox[
   Map[ButtonBox[ToString û #,
         ButtonFunction ¶ newPress[#],
         ButtonEvaluator Æ Automatic,
         Background Æ RGBColor[_]]&,
    buttonNames]
   ]], Active Æ True, TextAlignment Æ Center,
  FontFamily Æ "Courier", FontSize Æ 20, FontWeight Æ "Bold",
 CellTags Æ "newButtons"];

RGBPlaces = Position[newInterface, RGBColor[_]];
```

The new buttons use a new press function (otherwise they'd control the user interface simulation shown earlier in this paper!). The code is much as before, except that a loop assigns colours to each button: red for buttons that change state, and light gray if they do not change state.

```
newPress[theButton_] := Module[{nb = ButtonNotebook[]},
  state = device⟦buttonNames ' theButton, stateNames ' state⟧;
  NotebookFind[nb, "newButtons", All, CellTags];
  Do[newInterface = ReplacePart[newInterface,
     If[state === device⟦i, stateNames ' state⟧,
      RGBColor[0.9, 0.9, 0.9], RGBColor[1, 0, 0]
     ], RGBPlaces⟦i⟧], {i, numberOfButtons}];
  NotebookWrite[nb, newInterface];
  NotebookFind[nb, "newDisplay", All, CellTags];
  SelectionMove[nb, All, CellContents];
  NotebookWrite[nb, Cell[ToString û state]]
 ]
```

It is possible that the user interface simulation described above has been used (which can only happen if this paper is run in a *Mathematica* session, rather than just being read on paper), so at this point we don't know what the actual state of the device should be, and so we don't know what colour the buttons should be. The easiest thing is to press any button manually, so the code will update the state and set the button colours correctly.

clock quickDefrost time clear power

quickDefrost

We hope that the highlighting of a button affects whether a button is pressed; a button should be pressed only if it is highlighted. More generally, changing the physical design of the user interface will affect how likely a button is pressed. For example, if a button is made bigger and has a more attractive appearance it would be used more. The question is, how would this affect the user's ability to perform tasks? To help answer this design question, we can use the mean first passage function to work out the expected time of performing a task as a function of the proportion of time a button is used.

As an example, we construct two matrices: `nonTime` (which represents a user who never presses the `time` button) and `onlyTime` (which represents a user who only presses the `time` button).

```
onlyTime =
 nonTime = Table[0, {numberOfStates}, {numberOfStates}];

Do[If[buttonNames[[b]] =!= time, nonTime[[i,
    stateNames ' device[[b, i]]]] += 1 / (numberOfButtons - 1)],
  {b, numberOfButtons}, {i, numberOfStates}];

Do[If[buttonNames[[b]] == time,
  onlyTime[[i, stateNames ' device[[b, i]]]] += 1],
  {b, numberOfButtons}, {i, numberOfStates}];
```

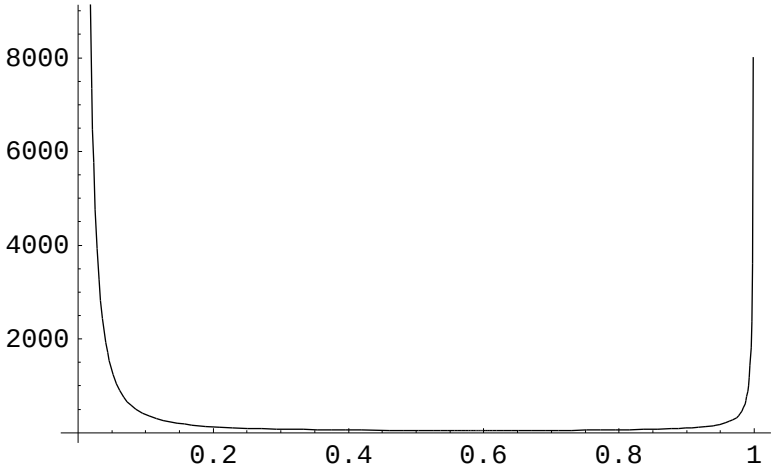
The `nonTime` matrix has zeros where a state transition can only happen by pressing the `time` button, whereas the `onlyTime` matrix has ones where the `time` button works. Where the `onlyTime` matrix is 1, the `nonTime` matrix must be zero. Printing the two matrices (below) makes things clearer!

```
Print[N@nonTime // TraditionalForm, "    ",
 onlyTime // TraditionalForm]
```

$$\begin{pmatrix} 0.75 & 0.25 & 0 & 0 & 0 & 0 \\ 0.5 & 0.5 & 0 & 0 & 0 & 0 \\ 0.5 & 0.25 & 0 & 0 & 0.25 & 0 \\ 0.5 & 0.25 & 0 & 0 & 0 & 0.25 \\ 0.5 & 0.25 & 0 & 0 & 0.25 & 0 \\ 0.5 & 0.25 & 0 & 0 & 0 & 0.25 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

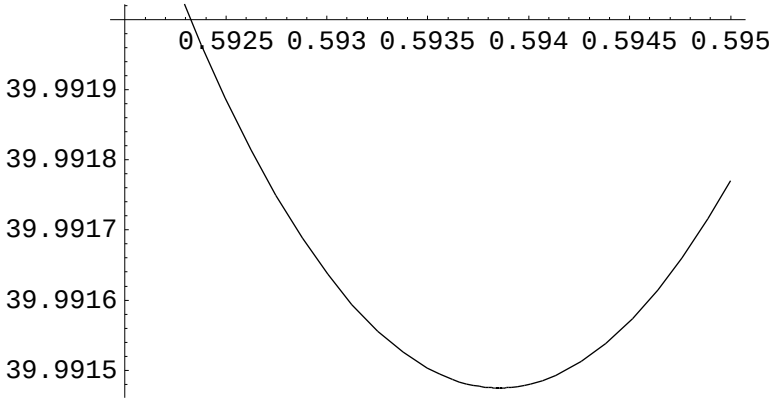
We can then plot the performance of a user whose behaviour is represented by a linear combination of these two matrices.

```
Plot[
 meanFirstPassage[(1 - k) nonTime + k onlyTime, start, goal],
 {k, .001, .999}];
```



We will skip the details, but by plotting the graph on a decreasing interval, we can find a narrower range of k that gives the best user performance:

```
Plot[
 meanFirstPassage[(1 - k) nonTime + k onlyTime, start, goal],
 {k, .592, .595}];
```



This is saying that, for the given task and other things being equal, making the `time` button be used 60% of the time (three times more likely than a 'fair' use of 20%) will make the device easier to use. In actual design, we should consider all possible tasks the device is intended to support, and we should attach weights to each task (e.g., quick defrost is less important, or done less often, than cooking at power level 1); then, we could calculate the optimal 'size' (relative frequency of use) for each button.

Automatic (and hence correct) help

There is a legal requirement that descriptions of products correspond with the products themselves: under the Sale of Goods Act 1979 (as amended by the Sale and Supply of Goods Act 1994 and the Sale of Goods (Amendment) Act 1994) products should be 'fit for purpose' and should correspond with the description of them. Thus it is the (UK) law that user manuals are correct — or, if we take a weaker view, that the manufacturer at least *knows* what the correct description is, so that some appropriate description, but truthful, can be written for the user.

Although our device definition is very basic, it can be used to generate quite useful help for the user or for technical authors. We now define a function `help` that explains the shortest path (the least number of button presses) to get from any state to any state. The definitions given below can be adapted straight-forwardly to provide clearer help if 'buttons' aren't actually pressed (maybe they are knobs that have to be twisted).

```
whichButton[s_, f_] :=
 Do[If[device[[i, s]] === stateNames[[f]],
   Print["  Press ", buttonNames[[i]]]],
  {i, Length@buttonNames}];

help[s_, s_] :=
 Print["Nothing to do."];

help[s_, f_] :=
 Module[{p = ShortestPath[Graph[randomUser, {}], s, f]},
  Do[whichButton[p[[i]], p[[i + 1]]], {i, Length@p - 1}]]
```

The device might have an interactive feature, so pressing a button gives help, perhaps showing it in a display panel. If so, it might be defined partly as follows — making use of the current state:

```
 help[doWhat_] := help[state, doWhat]
```

Users may wish to ask (and get answered!) questions such as, "I pressed something, but I expected such-and-such; what should I have done?" Thimbleby & Addison (1996) discuss how to supply answers to such "intelligent help" questions.

We can use the `help` function to generate an entire user manual. A short function tells us how to get from one state to another:

```
explain[i_, j_] := (Print["To get from the device showing ",
   stateNames[[i]], " to showing ", stateNames[[j]], ":"];
  help[i, j])
```

And here is a small part of the manual:

```
explain[start, goal]
```

To get from the device showing power1 to showing power2:

Press time

Press power

Ideally one would write more sophisticated routines to generate better natural language, rather than the simplistic ones demonstrated here. In particular, straight-forward parametrisation of the routines would allow *equivalent* manuals to be generated in any appropriate language.

If we developed a typographical style for user manuals, then all devices processed would be able to use that style (compare this idea with the tabular typesetting of the device specification shown earlier). Also, one can generate HTML manuals for the World Wide Web, and then the user can also follow hypertext links to help understand the workings of the device.

The entire manual can be printed with the following *Mathematica* code:

```
Do[If[i ≠ j, explain[i, j]], {i, Length@stateNames},
 {j, Length@stateNames}];
```

This doesn't provide a particularly easy read (certainly not *all* of it!), but it is a complete and correct manual that a technical author could work from. However, for many devices, including this microwave cooker, a user's tasks won't be so-much to get from a known state to another state, but simply to get to the desired state, *regardless* of the initial state. We will now generate a manual for this sort of use.

To represent a device in an unknown state, we represent its possible states as a set, and we define a function to find out what set of states the device will be in after a given sequence of button presses:

```
StateSet[initialStates_, presses_] :=
 If[presses == {}, initialStates,
  StateSet[Union[Map[
     stateNames ' device[[First[presses], #]]&, initialStates]],
   Rest[presses]]]
```

A breadth-first search can then be used to look for unique states:

```
NewManual[explain_] :=
 Module[{allStates = Range[numberOfStates], goals, queue},
  goals = allStates;
  Search[seq_] :=
   Do[Module[{p = Append[seq, b], g},
     g = StateSet[allStates, p];
     If[Length[g] == 1 && MemberQ[goals, g[[1]]],
      explain[g[[1]], p];
      goals = DeleteCases[goals, g[[1]]]];
     AppendTo[queue, p]],
    {b, numberOfButtons}];
  Search[queue = {}];
  While[goals != {},
   Search[First[queue]];
   queue = Rest[queue]]
 ]
```

Then, by defining some routines to explain things in (for instance!) English, we can print out the sequences of button presses to get to each state. We now have the user manual that tells a user how to do anything regardless of what the device is doing to start with. Notice how short it is; perhaps because of its brevity, as we shall soon see, we can get some interesting design insights straight from it.

```
Print[
 "Whatever the device is doing, you can always get it to"];

SayList[{s_}] := s<>".";
SayList[{s_, t_}] := s<>", then "<>SayList[{t}];
SayList[{s_, t__}] := s<>", "<>SayList[{t}];

English[state_, actions_] := Print["  ", stateNames[[state]],
  " by pressing ", SayList[ToString/@buttonNames[[actions]]]];

NewManual[English]
```

Whatever the device is doing, you can always get it to

clock by pressing clock.

quickDefrost by pressing quickDefrost.

timer1 by pressing clock, then time.

timer2 by pressing clock, time, then time.

power1 by pressing clock, time, then power.

power2 by pressing clock, time, time, then power.

Looking at these instructions, it looks like the `clock` button ought to have been called `reset`. If so, note that you can still get to state `quickDefrost` by pressing `reset` (i.e., `clock`) first, then the `quickDefrost` button. Also, we might think that if such a manual is 'good,' what would a device look like that this manual was the *complete* explanation for? To find out, all we need to do is change the `English` routine to one that goes back to the device specification and sees what parts of it are used, and which are not.

```
d = device;

checkUsed[s_, actions_] :=
 Module[{i, states = Range[numberOfStates]},
  For[
   i = 1, {s} π states, states = StateSet[states, {actions⟦i++⟧}],
   Scan[(d⟦actions⟦i⟧, #⟧ = "—")&, states]]];

NewManual[checkUsed];
neatTable["Actions that weren't needed for the manual",
 d, stateNames, buttonNames]
```

Buttons	— States —					
	clock	**quickDefrost**	**timer1**	**timer2**	**power1**	**power2**
clock	—	—	—	—	—	—
quickDefrost	—	—	—	—	—	—
time	—	timer1	—	timer1	timer2	timer1
clear	clock	clock	clock	clock	clock	clock
power	clock	quickDefrost	—	—	power1	power2

Actions that weren' t needed for the manual

We can look closely at the non-blank entries in this table: these are the parts of the specification that the user manual did not require. Amongst other comments: the `clear` button doesn't seem to be helping much! (Probably Sharp's specification does not say what `clear` really does: it probably clears a numerical timer setting that he wasn't interested in.) Nevertheless, our generating a manual and then automatically going back to the specification has exposed some potential bad design. If this sort of manual is a good idea, then the `clear` button as presently defined is a design feature that needs better justification.

Many other sorts of manuals can be generated too, and by creating them using *Mathematica* or some other such system systematically we can *guarantee* their correctness. We can also use the technique of going back from a good manual to reappraise the specification. After all, if we have a good user manual, then the bits in the specification that aren't apparently needed are immediately suspicious features!

Elsewhere we discuss how the technical author's editing (starting from a correct manual) can be effectively managed, even as the device specification changes (Thimbleby & Ladkin, 1995). It is possible (but requires rather a lot of technical detail beyond the scope of this paper) to do something similar in *Mathematica*: the output of the manual generation can be written to a notebook, where the technical author can freely edit it (as a normal *Mathematica* document) and so make the user manual as readable as desired.

Mathematica allows 'cells' (i.e., manual paragraphs) to be tagged; using the tags, each paragraph can be uniquely identified, even though the technical author has edited them. Now, if the device specification changes, the notebook can be re-read, and a report automatically made of any cells whose original generated text has changed (or is new or has been deleted). This report can be automatically interleaved back into the manual, so that the technical author could more easily associate the comments with the affected parts of the manual.

The technical author can also point out peculiar features, or ones that are hard to explain: *Mathematica* could then track these suitably-flagged comments back to the offending parts of the specification, much like we did above (for instance, the technical author's comments would end up in the specification table, instead of the '—' dashes).

■ Conclusions

The development method described in this paper is very powerful, and with a system such as *Mathematica* it is also very easy to do. With *Mathematica* or with bespoke design packages, all the code could be concealed from designers: this paper — because it is explicit — gives an unnecessarily technical feel to the approach. The method is not limited to finite state machines (as might be supposed); Thimbleby & Ladkin (1997) discuss generating user manuals for quite complex systems, such as parts of the A320 fly-by-wire airplane, where we use a logic-based approach.

The *Mathematica* code shown in this paper will work with other devices, by making only the appropriate changes to the device specification. This paper, then, is itself a complete gadget-design package — *everything* discussed in this paper is explicitly and fully defined — and one is surprised that more devices are not designed in this way, rather than by using superficial tools that emphasise looks against specification.

Mathematica could be accused of being esoteric (it does have complexities this paper avoided); our further work is using Java to allow the user interface of the development environment to be put on the World Wide Web, and for designers anywhere in the world to write Java applets that can be analysed and simulated on the site. With world-wide use of simulations, one would be able to obtain global empirical statistics of device use. We also hope to promote good practice in user interface design.

■ References

P. Cairns, M. Jones & H. W. Thimbleby, 1998, "Reusable Usability Analysis with Markov Models," working paper (available from the authors).

J. Sharp, 1998, *Interaction Design for Electronic Products using Virtual Simulations*, PhD thesis, Brunel University.

N. Stanton, ed., 1998, *Human Factors in Consumer Products*, Taylor & Francis.

H. W. Thimbleby & M. A. Addison, 1994, "Manuals as Structured Programs," in G. Cockton, S. W. Draper and G. R. S. Weir eds., BCS Conference HCI'94, *People and Computers*, **IX**, pp67–79, Cambridge University Press.

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

H. W. Thimbleby & P. B. Ladkin, 1995, "A Proper Explanation When You Need One," in M. A. R. Kirby, A. J. Dix & J. E. Finlay eds., BCS Conference HCI'95, *People and Computers*, **X**, pp107–118, Cambridge University Press.

H. W. Thimbleby & P. B. Ladkin, 1997, "From Logic to Manuals Again," *IEE Proceedings Software Engineering*, **144**(3), pp185–192.

H. W. Thimbleby & I. H. Witten, 1993, "User Modelling as Machine Identification: New Design Methods for HCI," in D. Hix & R. Hartson eds., *Advances in Human Computer Interaction*, **IV**, pp58–86, Ablex.

H. W. Thimbleby, 1994, "Formulating Usability," *ACM SIGCHI Bulletin*, **26**(2), pp59–64.

H. W. Thimbleby, 1997, "Design for a Fax," *Personal Technologies*, **1**(2), pp101–117.

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

Acknowledgements

This paper is a summary of collaborative effort. The author is grateful to Ann Blandford, Paul Cairns, Matt Jones, Peter Ladkin, Gary Marsden and Ian Witten. A preliminary version of this paper was presented at the Institution of Electrical Engineers seminar, *Living life to the full with personal technologies*, London, 3 June 1998; and was published in the seminar digest (IEE Digest 98/268, pp4/1–4/9).