Formal Methods in HCI: Moving Towards an Engineering Approach

Alan J. Dix*
H.C.I. Group, Department of Computer Science,
University of York, Heslington, United Kingdom, YO1 5DD.
E-mail: alan@minster.york.ac.uk
Tel: (0904) 432778

February 23, 1993

Abstract

The author and others have been studying the interplay of formal methods and HCI for several years. In particular, much of this work has centred on the design of formal models of interactive systems which can be used to formalise properties of usability. Although this work has been successful, it requires quite a high level of mathematical sophistication and is thus hard to 'give away' to the practitioner. This paper will describe two methods which have their roots in formal analysis, but which do not require great formal expertise. Such methods can be thought of as operating at an 'engineering level' as they have to some extent pre-packaged the results and insights of more sophisticated analysis into a form more readily applied to practical problems.

1 Introduction

For many years I have worked on the interplay between formal methods and human–computer interaction. This area of research is particularly associated with (present and past) workers from York and there are now several books on aspects of this area. Various books and papers are described in the annotated bibliography at the end of this paper. In particular, most of the material in this paper can be found in an expanded form in a HCI text book which I recently co-authored [1], and in my previous monograph [2].

Much of my work has concerned the development of formal models of interactive systems. Of these, the PIE model, described in Section 2, is one of the oldest and most well known. This model was designed to express generic properties at the level of WYSIWYG (what you see is what you get) or the meaning of undo. Such models can be used in two principle ways:

(i) They give new insight into general problems as the properties of the problem domain are analysed.

(ii) They can be used as part of a formal development process to constrain the design of specific systems.

The models have been successfully used on both counts. unfortunately, they require a considerable degree of expertise in both formal methods and human factors. If we assume that the use of formal methods in software engineering continues to gain favour and also that the importance of good interface design continues to be regarded as important, then it may be that such combinations of skills become more common. However, whether or not this comes to pass, such skills are not at present the norm.

Considering point (i) this is not a problem. An expert analyst can gain new insight from the models and then either expound this knowledge in the context of the models, or even recast

*Alan Dix is funded by a SERC Advanced Fellowship B/89/ITA/220.

it completely in informal terms. Indeed, this has been the pattern of much of my own work! The reader or listener need not have the same level of familiarity with the model to follow the arguments, even where the model is used in the exposition.

However, point (ii), the use in a formal development process, requires that such an expert be available on the design team. Furthermore, the burden of proof can become excessive. The real expert is able to focus on critical areas, but the proficient practitioner may become lost in the morass of detail. This problem is not unique to the use of formal methods in interface design, but is a general problem in all applications of formal methods within software engineering.

One could wait for research in software engineering to attack the general problem, possibly with the aid of complex proof assistants and toolsets. But, like waiting for the effects of education to filter through, this is at best a long term hope.

Happily, there are developments which make use of methods derived from these models, yet which require less formal expertise on the part of the designer. These I will call 'engineering' level techniques. In these, the theoretical work is effectively packaged in the same way that in civil engineering the theoretical analysis of materials and soil mechanics is packaged up into tables, structural analysis programs and rules of thumb.

I will describe two such 'engineering' level approaches. The first, in Section 3, shows how properties similar to those defined over formal interaction models can be applied to dialogue descriptions. This has the advantage that dialogue descriptions are often produced for specification or prototyping purposes and that the analyses are simpler and can often be automated.

The second approach is status/event analysis This uses a distinction drawn from formal modelling work, but which can give insight into general and specific interface design with little reference to its formal roots. In particular, I will demonstrate how status/event diagrams can be used to analyse the usability of interface widgets.

2 Formal models of interaction

It's easy to say that a system is *WYSIWYG*, is *consistent* or has a universal *undo* facility, but how do you know? Is there any way we can take a system and say 'yes' it is or 'no' it is not WYSIWYG? It was just for this purpose that formal models of interaction were developed. One of the oldest such models is the *PIE* [5]. This is a general purpose model intended to apply to a wide range of systems. More specific models can address particular design areas, for example, there are models for windowing systems, models for describing temporal behaviour and models for dealing with non-determinism[2]. At the opposite extreme from the PIE model are specifications of specific systems, for example, Sufrin's specification of a text editor [6]. These of course can only be used to reason about the one system (although the specifier may draw general lessons).

The rest of this section will concentrate on the PIE model and a few of the usability properties which it can be used to describe. We will look at two broad classes of properties. Those concerning what you can *see* of the system: predictability and observability, and those concerning what you can *do* to the system: reachability and undo. The concept of WYSIWYG is partly captured within the PIE's *predictability* property, but like all usability properties, we find that the formal statement is only a partial expression of the real property we desire. Nevertheless, some of the formal principles are necessary for usability, any system which breaks them is bound to have problems. The formal principles form a 'safety net' to prevent some of the worst mistakes in an interactive system. Formal analysis is thus an aid to, but *not* a substitute for, good design.

Concepts underlying the PIE model

The PIE model is a *black-box model*. It does not try to represent the internal architecture and construction of a computer system, but instead describes it purely in terms of its inputs from the user and outputs to the user. For a simple single-user system typical inputs would be from keyboard and mouse, and outputs would be the computer's display screen and the eventual printed output (Figure 1).

The difference between the ephemeral *display* of a system, and the permanent *result* is central to the PIE model. We will call the set of possible displays D and the set of possible results R. In order to express principles of observability, we will want to talk about the relation between display

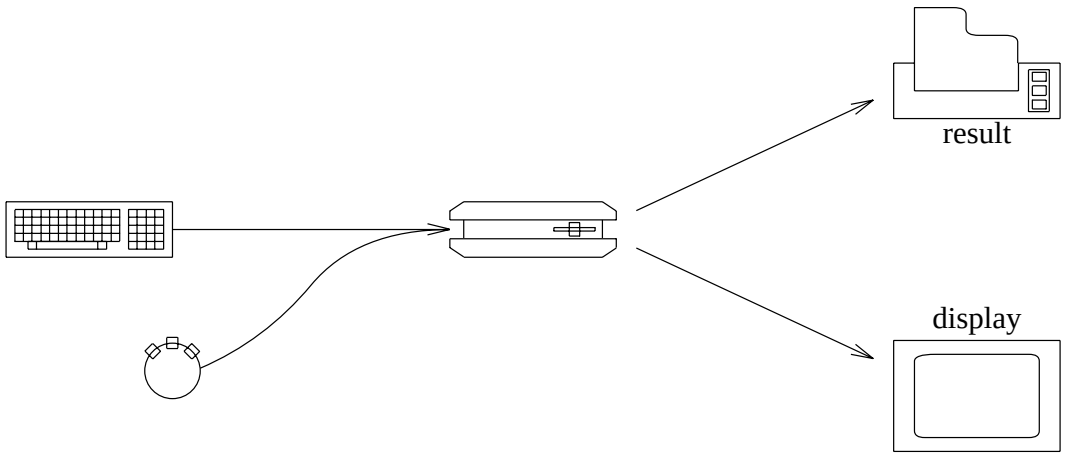

Figure 1: Inputs and outputs of single user system

and result. Basically, can we determine the result (what you will get) from the display (what you see)?

More formally ...

For a formal statement of predictability it helps (but is not essential) to talk about the internal state of the system. This does not counter our claim to have a black-box model. First, the state we define will be opaque, we will not look at its structure, merely postulate it is there. Second, the state we will be discussing is not the actual state of the system, but an idealisation of it. It will be the minimal state required to account for the future *external behaviour*. We will call this the *effect* (E). Functions *display* and *result* obtain the current outputs from this minimal state:

$$display : E \rightarrow D$$
$$result : E \rightarrow R$$

The current display would be literally what is now visible. The current result is actually not what *is* available, but what the result would be if the interaction were finished. For example, with a word-processor it is the pages that would be obtained if one printed the current state of the document.

A single user action we will call a *command* (from a set C). The history of all the user's commands is called the *program* ($P = \text{seq}\, C$), and the current effect can be calculated from this history using and *interpretation function*:

$$I : P \rightarrow E$$

Arguably the input history would be better labeled H, but then the PIE model would loose its acronym!

If we put together all the bits, we obtain a diagram of sets and functions (Figure 2), which looks rather like the original illustration.

In principle, one can express all the properties one wants in terms of the interpretation function I. However, this often means expressing properties quantified over all possible past histories. To make some of the properties easier to express, we will also use a state transition function *doit*:

$$doit : E \times P \rightarrow E$$

The function *doit* takes the present state e and some user commands p, and gives the new state after the user has entered the commands $doit(e, p)$. It is related to the interpretation function I by the following:

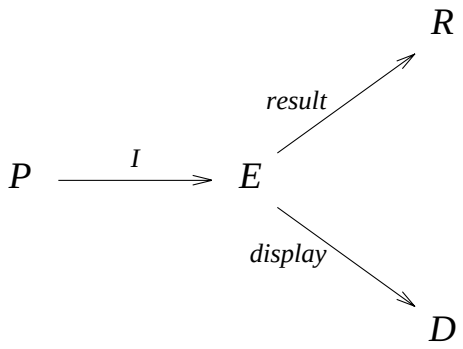

Figure 2: The PIE model

$$
\begin{array}{lcl}
doit(I(p), q) & = & I(p \frown q) \\
doit(doit(e, p), q) & = & doit(e, p \frown q)
\end{array}
$$

Now the PIE diagram can be read at different levels of abstraction. On can take a direct analogy with Figure 1. The commands set C is the keystrokes and mouse clicks, the display set D is the physical display, and the result R is the printed output.

$$
\begin{array}{lcl}
C & = & \{\ \text{`a'}, \text{`b'}, \ldots, \text{`0'}, \text{`1'}, \ldots, \text{`*'}, \text{`\&'}, \ldots\} \\
D & = & Pixel\_coord \rightarrow RGB\_value \\
R & = & \text{ink on paper}
\end{array}
$$

This is a physical/lexical level of interpretation. One can produce a similar mapping for any system, in terms of the raw physical inputs and outputs. It is often more useful to apply the model at the logical level. Here the user commands are higher level actions such as 'select bold font' which may be invoked by several keystrokes and/or mouse actions. Similarly, we can describe the screen at a logical level in terms of windows, buttons, fields etc. Also, for some purposes, rather than dealing with the final physical result, we may regard say the document on disk as the result.

The power of the PIE model is that it can be applied at many levels of abstraction. Some properties may only be valid at one level, but many should be true at all levels of system description. It is even possible, to apply the PIE model just *within* the user, in the sense that the commands are the user's intended actions and the display, the perceived response.

When applying the PIE model at different levels it is possible to map between the levels. This leads to *level conformance* properties, which say, for example, that the changes one sees at the interface level should correspond to similar changes at the level of application objects.

Observability and predictability

The WYSIWYG is clearly related to what can be inferred from the display (what you see). Harold Thimbleby has pointed out that WYSIWYG can be given two interpretations [4]. One is what you see is what you *will get* at the printer. This corresponds to how well you can determine the result from the display. The second interpretation is what you see is what you *have got* in the system. For this we will ask what the display can tell us about the effect. These can both be thought of as *observability* principles.

A related issue is *predictability*. Imagine you have been using a drawing package and in the middle you get thirsty and go to get a cup of tea. On returning, you are faced with the screen – do you know what to do next. If there are two shapes one on top of the other, the graphics package may interpret mouse clicks as operating on the 'top' shape. However, there may be no visual indication of which is topmost. That is the screen image does not tell you what the effect

of your actions will be, you need to remember how you got there, your command history. This has been called the '*gone away for a cup of tea problem*'.

In fact the state of the system determines the effects of any future commands, so if we have a system which is observable in the sense that the display determines the state, it is also predictable. Predictability is a special case of observability.

Let's have a go at formalising these properties. To say that we can determine the result from the display is to say that there exists a function $transparent_R$ from displays to results.

$$\exists\, transparent_R : D \rightarrow R \\ \quad \bullet\ \forall\, e \in E \bullet transparent_R(display(e)) = result(e)$$

Of course, there is no good having any old function from the display to the result, the second half of the above says that the function gives us exactly the result we would get from the system. This property is a good first cut at *observability*, but will in fact turn out to be too strong. For now call it *result transparency*. It says that the display contains at least as much information as the result. However, it may also contain additional information about the interactive state of the system. For example, you will observe the current cursor position, but this has no bearing on the printed document.

So you know what will happen if you hit the print button *now*. Refreshed from your cup of tea, you return to work. You press a function key which, unbeknown to you, is bound to a macro intended for an entirely different application. The screen rolls, the disk whirs and to your horror your document and the entire disk contents are trashed. You leave the computer and go for another drink ...not of tea.

A stronger condition can be obtained if we demand that the system state can be observed from the display:

$$\exists\, transparent_E : D \rightarrow E \\ \quad \bullet\ \forall\, e \in E \bullet transparent_E(display(e)) = e$$

We can regard this as an initial attempt at *predictability*, but, it will again turn out to be too strong, so instead we will call the above property simply *transparency*.

What would it mean for a system to be transparent in one of these senses. Well, if the system were result transparent, when we come back from our cup of tea, we can look at the display and then work out in our head (using $transparent_R$) exactly what the printed drawing would look like. Of course, whether we could do this in our heads is another matter. For most drawing packages the function would be to simply ignore the menus and 'photocopy' the screen.

Simple transparency is stronger still. It would say that there is nothing in the state of the system that can not be inferred from the display. If there are any modes, then these must have a visual indication, if there are any differences in behaviour between the displayed shapes, then there must be some corresponding visual difference. Even forgetting the formal principles, this is a strong and useful design heuristic.

Unfortunately, these principles are both rather too strong. If we imagine a word-processor rather than a drawing package, the contents of the display will be only a bit of the document. Clearly, we cannot infer the contents of the rest of the document (and hence the printed result) from the display. Similarly, to give a visual indication of say object grouping within a complex drawing package might be impossible (and this can cause the user problems).

On the other hand, one could regard the transparency properties as being too *weak*. The function $transparent_R$ represents the reasoning the user would have to do to predict the result from the current display. That such a function exists is no guarantee that the user can perform the calculation – it may involve the translation of Ancient Egyptian hieroglyphics.[1]

When faced with a document on a word-processor, the user can simply scroll the display up and down to find out what is there. That is, you cannot see from the current display everything about the system, but you can find out. The process by which the user explores the current state of the system is called a *strategy*. The formalisation of a strategy is quite complex, even ignoring cognitive limitations. These strategies will differ from user to user, but the documentation of a system should tell the user how to get at pertinent information. For example, how to tell what

[1] Which is precisely how non-English speakers feel when faced with many menu driven interfaces.

objects in the drawing tools are grouped. This will map out a set of effective strategies with which the user can work.

One can use the idea of a strategy to formulate more effective observability and predictability properties. Indeed, one can go further still and generate models and properties which take into account aspects of user attention, and issues like keyboard buffers. The books referred to in the bibliography deal with such extensions.

Principles of predictability do not stand on their own even if you had known what was bound to the function key, you might still have hit it by accident, or simply forgotten. Other protective principles like *commensurate effort* need to be applied [4]. Also, although it is difficult to formalise completely, one prefers a system which behaves in most respects like the transparency principles, rather than requiring complicated searching to discover information. That is a sort of commensurate effort for observation.

Reachability and undo

In a commercial program debugger, there is a window listing all the variables. If a variable is a complex structure, then hitting the 'insert' key while the cursor is over the variable will expand the variable showing all its fields. If you only want a few of the fields to be displayed, you can move the cursor over the unwanted fields and type the 'delete' key and the field is removed. These operations can be repeated over complex hierarchical structures. If you remove a field and then wish you hadn't, you can always press insert' again over the main variable and all the fields will be re-displayed. Even this breaks somewhat the principle of *commensurate* effort, but worse is to come. The 'delete' key also works for top level variables, but once one of these is removed from the display there is *nothing* you can do to get it back, short of exiting the debugger and re-running it from scratch.

A principle which stops this type of behaviour is *reachability*. A system is reachable if from any state the system is in, you can get to any other state. The formal statement of this is as follows:

$$\forall e, e' \in E \bullet \exists p \in P \bullet doit(e, p) = e'$$

Unlike the predictability principles, there are no awkward caveats. The only problem is that, if anything, it is too weak. For instance, a word-processor could have a delete key, but no way to move the cursor about, you always type at the end of the document. Now you can, of course, get from any document state to any other, you simply delete the whole text and retype what you want. However, if you had just typed in a whole letter then noticed a mistake on the first line, you would not be pleased! So, ideally one wants an independent idea of 'distance' between states and make the difficulty of the path between them commensurate with the distance – small changes should be easy. Despite this, the principle on its own would have been strong enough to prevent the behaviour of the debugger!

One special case of reachability is when the state you want to get to is the one you have just been in. That is, *undo*. We expect undo to be easy, and ideally have a single undo button that will always undo the effect of the last command. We can state this requirement very easily:

$$\forall c \in C \bullet doit(e, c \frown undo) = e$$

This says exactly what we wanted. We start in a state e. We then do any command c and follow it by the special command *undo*. The state is then the same as we began in.

Stop! Before clapping ourselves on the back for so clearly defining undo, we should check that this requirement for undo is consistent. Indeed, it is consistent – so long as there are at most two states. That is, the above undo requirement is only possible for systems which *do* virtually nothing! The reason for this is that *undo* is itself a command and can undo itself. Take any state e and choose any command x. Let e_x be the state you get to after command x. That is $e_x = doit(e, x)$. Now we can apply the undo requirement to state e_x:

$$doit(e_x, undo) = doit(e, x \frown undo) = e$$

So, the *undo* command in state e_x gets us back to e. That is as expected. But what does *undo* do if we are in state e. Again we can employ the undo principle remembering that $e = doit(e_x, undo)$:

$$doit(e, undo) = doit(e_x, undo \frown undo) = e_x$$

This uses the undo principle when the command c is undo itself. However, our choice of command x was arbitrary, so if we had chosen another command say y we would have concluded that $doit(e, undo) = e_y$. This means that $e_x = e_y$, and in general anything we do from state e gets us to the same state. With a little more argument you can show that any command from this second state gets us back to the original one. So at very most we have two states, a toggle, with all the commands flipping back and forth between them. The only other alternative is that the system does nothing.

We won't go on to describe the details of better undo requirements, the interested reader can find that elsewhere. The basis of most workable undo systems is that *undo* is not just any old command, but is treated differently. The simplest fix to the above undo principle is to restrict the commands to anything *except undo*!

The lesson from the above is clear. It is easy to say you want something which sounds quite reasonable. A formal description of the requirement may well reveal that, as in the case of undo, it is inconsistent – that is *no* system could be built which satisfies the requirement.

Summary – formal modelling

The example of undo shows how useful formal models can be as tools for understanding. The specification we originally gave sounded good enough, but was inconsistent. If we had tried to build a system having such an undo, we would either fail, or *think* we had succeeded. In the former case, we might keep fruitlessly trying to build a system with a single universally applicable undo button. In the latter, we might delude ourselves into thinking this was what we had, only to discover (after selling the system!) that there were cases where it failed.

However, it is also clear that the job of verifying that a large interactive system is reachable or predictable may be very difficult.

3 Dialogue Analysis

The difficulty about proving properties of systems is that the state is very complex. For example, the state of a word processor will contain information such as:

Screen:	edit screen
Text:	"to be or not to be, ..."
Menu:	file menu displayed
Cursor:	at the 7th character line 12

To be able to prove things about such a state, we need to reason about numbers and text as well as mode indicators such as **Screen** and **Menu**. The number of possible texts and cursor positions is infinite, or even if we take into account system limits *very large*. This means we have to reason symbolically – heavy mathematics!

Dialogue descriptions usually limit themselves to the finite attributes of the state. Those which have a major effect on the allowable sequences of user actions. They are thus instantly more amenable to automated analysis (we can sometimes simply try all cases). Furthermore, dialogue descriptions are often used as part of design anyway, thus we may be able to take an existing product of the design process and obtain instant added value.

3.1 Notations

There are a large number of different dialogue notations. Some use diagrammatic representations of the dialogue (see below) and others use textual representations (such as the use of grammars or production rules).

Of the diagrammatic techniques, *state transition network* (STNs) are most heavily used. (But even they come in several variants.) We will base our discussion primarily on STNs, but other notations could equally be used.

State transition nets consist of two elements:

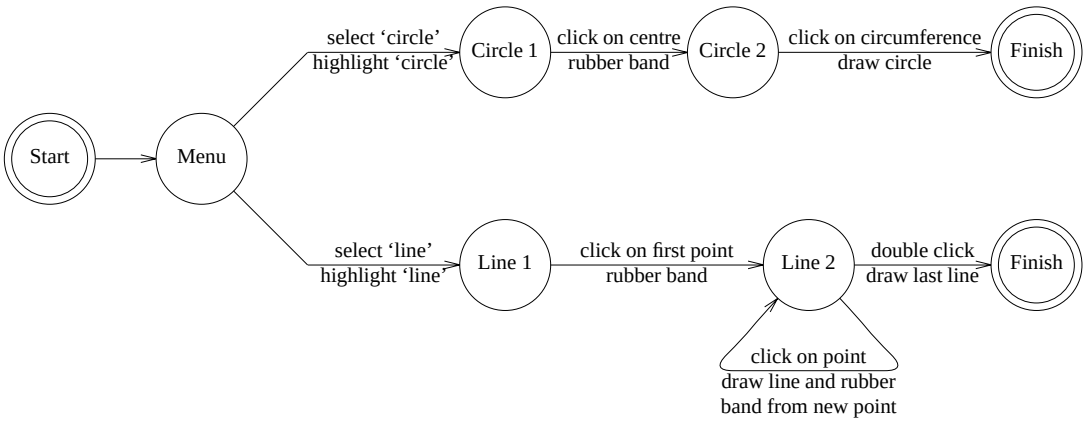

Figure 3: State transition network for menu driven drawing tool

circles – denoting the states of the dialogue

arcs – between the circles, denoting the user actions/events

Figure 3 shows a STN describing a portion of the dialogue of a simple drawing tool. The arcs are also labelled with the feedback or system response resulting from the user's actions. Note how cramped the arcs get – obviously a lot is happening at each event.

The STN for a full system would usually be enormous. To manage the complexity, STNs are often described hierarchically. For example, Figure 4 shows the higher level dialogue for the drawing tool, selecting between several sub-menus. The menu in Figure 3 corresponds to the graphics sub-menu. Each of the sub-menus would have similar STNs describing them.

The hierarchical decomposition in this diagram is of states. Single states in the high-level diagram correspond to an entire low-level STN. There are other possibilities for hierarchical decomposition, for example, augmented transition networks allow both user actions and system responses to be decomposed into further STNs.

3.2 Why do people use dialogue notations?

I said that we were focusing on dialogue descriptions because they often 'come for free', a natural product of the design process. There are several reasons for this:

- The use of UIMS or UIDEs.
- For dialogue specification on paper.
- For rapid prototyping.

We'll look at these in turn.

UIMS

If we use a *User Interface Management System* (UIMS) or User Interface Development Environment (UIDE) this will usually include a formal description of the dialogue. This may be in the form of production rules, a grammar or even some graphical representation. Some of these representations, especially production rules, do not completely separate the dialogue from the underlying state. However, the conversion required is certainly far less work than generating the description from scratch *and* is guaranteed to be consistent with the actual system.

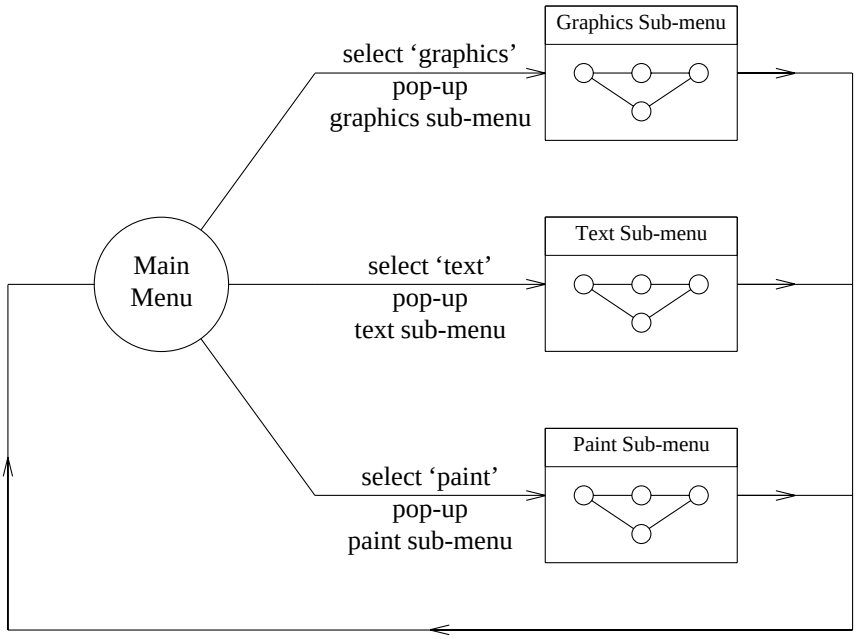

Figure 4: Hierarchical state transition network for complete drawing tool

Paper specification

A second reason for the use of dialogue descriptions is simply as a paper specification method, just as one might use data-flow diagrams for information systems or entity-relationship diagrams for database design. Several years ago I was working in a data processing department producing information systems under a forms-based transaction processing (TP) environment.

Programming a TP system is similar to many window systems, basically a stimulus-response model. Your program gets a screen full of data and must decide what to do with it. When it has processed that screen, it sends a fresh template to the user and then goes on to service a *different* terminal. Because of this form of programming, one cannot implicitly encode the dialogue within the program structure. So, for example, it is quite difficult to ensure that the user can only delete a record after it has been displayed.

To ease the problem of writing (relatively) complex dialogues under this regime, the author used flowcharts to describe the interaction with each user. Figure 5 shows a flowchart for a delete sub-dialogue similar to those used.

Note two things, despite surface similarities, there are important differences both from normal program flowcharts and from STNs.

First, note that a flowchart of the program implementing this dialogue would (because of the stimulus-response model) be tree-like. It would have to explicitly store the dialogue state and generally being totally incomprehensible *without* the corresponding dialogue description. Furthermore, the sorts of things one puts in the boxes of a dialogue flowchart are different from program flowcharts. For example, reading a record could be a complex activity, say searching through a file until the matching record is found. However, from the dialogue viewpoint this corresponds to a single system action.

Note also that although superficially like an STN, with boxes connected by arrows, the emphasis is rather different. The boxes represent system processes or user interactions, that is, the notation is event/process oriented rather than state/oriented. We will return to this status/event distinction in Section 4.

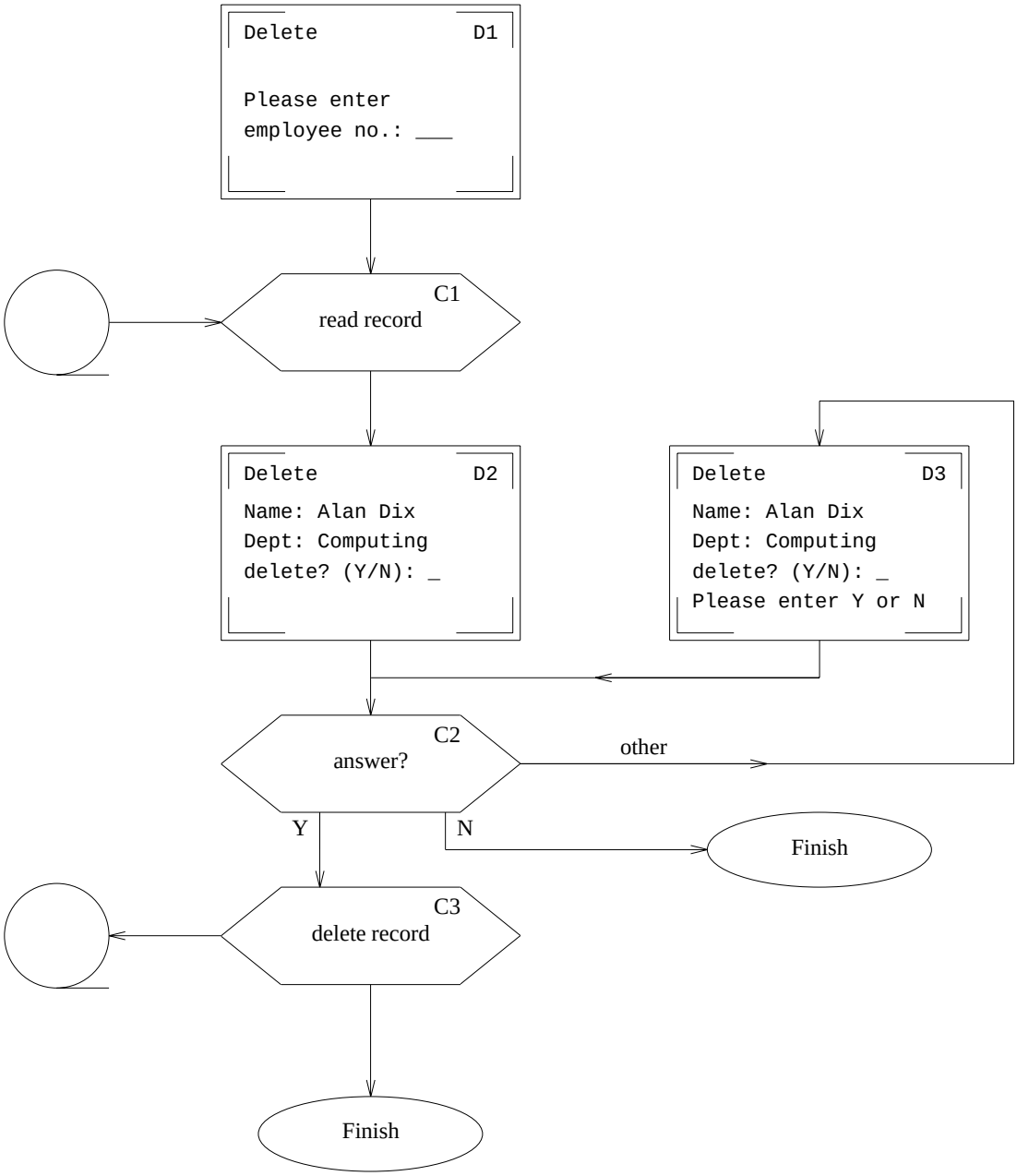

Figure 5: Flow chart of deletion sub-dialogue

In a different vein, formal notations are often criticised for the amount of work required. However, the author's experience counters this. The author used these diagrams and converted them, mechanically, but by hand into Cobol programs. Using this method I was able to produce within days, systems which had previously taken months to complete. Furthermore, changes could be accomplished within hours (no mean feat within such an environment!). Although, it might be nice to think this was due to superior programming skills (!), this could in no way account for an order of magnitude difference in productivity. That is, the adoption of a kind of formal notation did not waste valuable time, but instead made phenomenal time savings.

Figure 6: Hyperdoc

Prototyping

Dialogue descriptions can be used to drive prototyping tools or simulators. This is rather like the use with UIMS, but usually with a less extensive environment. One example of this is Heather Alexander's SPI notation (*Specification Prototyping and Interaction*) [11]. This uses a variant of CSP for the dialogue description and then has tools which allow one to 'run' the dialogue seeing the possible interaction paths.

Another support tool is Hyperdoc developed by Harold Thimbleby [12], shown in Figure 6. The screen shows part of the description for a JVC video-recorder. The top half of the screen is a drawing of the interface. The buttons on the drawing are active – the simulation runs when they are pressed. On the bottom left, we can see part of the dialogue description. This describes the transitions from the state 'playPause'. For example, if the user presses the 'Operate' button, the state will change to 'offTapein'.

In fact, this tool does more than simply simulate the dialogue, it can perform several forms of dialogue analysis.

3.3 Dialogue properties

Given a dialogue description, we can begin to look at what properties it satisfies. We look at these under two headings

action properties which describe local phenomena at a particular state. That is, concerning single actions.

state properties which concern the movement between states, which may encompass whole trains of actions.

After looking at these two kinds of properties, we will consider two examples of their use.

Action properties

There are several dialogue properties which are to do with local dialogue actions:

completeness – look at each state, is there an arc coming from that state for each possible user action? If not, what is the effect on the system if the user performs this action? This is a good way of checking for 'unforeseen circumstances'.

determinism – is the behaviour uniquely defined for each user action. In a simple STN this corresponds to checking that there is at most one arc labelled with each user action from a particular state. Non-determinism can be deliberate, corresponding to an application decision. However, it can be a mistake, and this is especially easy in complex hierarchical STNs, production rules systems etc. Automatic tools can help check for this.

consistency – does the same user action have a similar effect in different states? If not are these dialogue *modes* visibly different?

If we look back to Figure 3 we can check it for completeness. The action 'select-line' is not mentioned in either of the line states, but this is deliberate. The line option is assumed to be on a pop-up menu and so cannot occur except from the menu state. The remaining actions are then single and double clicks. What happens if we double click in either of the circle states? Is this signaled to the user as an error by a beep, simply ignored, does it do something odd (a feature!) or does it crash the program?

State properties

Another set of properties are more global, considering how easy or difficult it is to get from one state to another:

reachability – can you get anywhere from anywhere? That is, imagine you are at a particular dialogue state and you want to get to a different state. Is there a sequence of user actions which is guaranteed to get you there? In addition, we may want to ask just how complicated and long that sequence is.

reversibility – can you get to the previous state? Imagine you have just done an action, but wished you hadn't. This is a special case of reachability, but one which we expect to be especially easy – we all make mistakes. Note this is *not* undo – returning to a previous dialogue state does not in general reverse the semantic effect.

dangerous states – there are some states you don't want to get to. Does the system make it difficult to perform actions which take you to these dangerous states?

As an example, we can check the reversibility of the drawing tool (Figures 3 and 4). Imagine we want to reverse the effect of "select 'line'" from the graphics Menu state. We can perform three actions:

click – double click – select 'graphics'

These return us to the graphics pop-up menu. However, these will leave a vestigial circle on the display. That is, in this case, as we warned, reversing the dialogue is *not* undo.

Note also that this reachability for dialogue states is equivalent to the definition for full system states, but weaker. A system cannot be reachable in the PIE sense if it is not reachable at the dialogue level, but, like undo, dialogue reachability does not guarantee full reachability.

In graph theoretic terms, dialogue reachability is called *strong connectivity* and the Hyperdoc tool, described previously, is able to perform this analysis for the designer.

3.4 Example – Digital watch

3.4.1 User's documentation

A digital watch has a very limited interface – 3 buttons. These must control the watch display (time/calendar) a stopwatch mode and an alarm.

We only consider one of the buttons, button 'A', which is used to move between the four main modes: time/calendar, stopwatch, alarm setting and time setting.

Figure 7 shows a portion of the user instructions. It is a simple state transition network.

We can analyse this network. The time and alarm setting modes are dangerous states, we don't want to set the time by accident. These states are guarded – you have to hold the button down for two seconds. This button is very small and it is difficult to hold it down by accident.

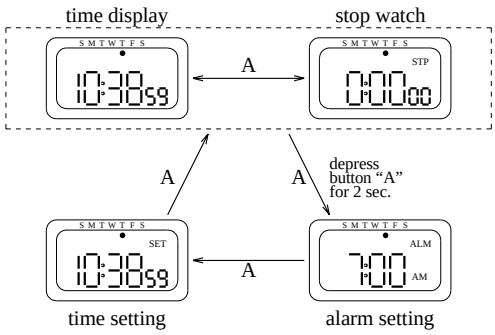

Figure 7: Instructions for digital watch

What about completeness? The idea of holding the button down suggests that we ought to distinguish the actions of depressing and releasing button 'A'. So, what do these actions do in the different modes?

Although the STN is incomplete this is acceptable for the user instructions so long as undocumented sequences of actions do not have a disastrous effect. However, the designer must investigate all possibilities to check this.

3.4.2 Designer's documentation

Extensive experiment eventually revealed the complete STN for the watch, shown in Figure 8. This includes for each state the effect of the three actions:

- depress A
- release A
- wait two seconds

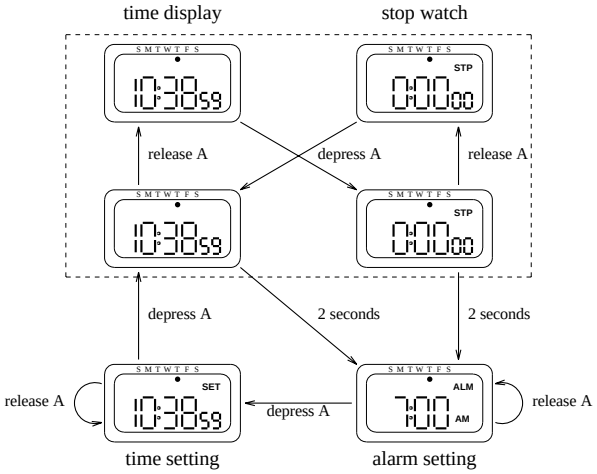

Figure 8: Design diagram for digital watch

Notice that this required the addition of two meta-stable versions of the time/calendar state and the stopwatch state. This is the sort of diagram that the designer would need to analyse and to pass on to the implementor.

The diagram looks fairly complex — and we've only looked at one button!

3.5 Example – Dangerous states

One of the word processors being used to prepare this document exhibits dangerous states. It has two main modes, the main mode where you edit the text, a menu and help screen from where you perform filing operations. You switch between these modes with the 'F1' key. In addition, from the menu you can exit the word processor by hitting the 'F2' key. These modes and the exit are shown in Figure 9.

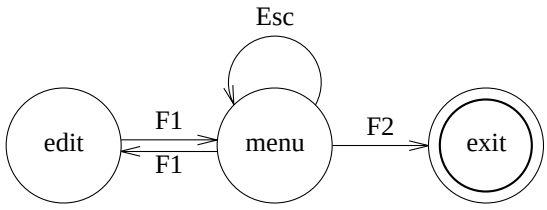

Figure 9: Main modes of text editor

If the text has been altered it is automatically saved upon exit. However, if you have altered the text, but then decide to abandon your edits, this automatic save can be turned off by hitting the escape key in the Menu mode. Subsequent edits will reset this and the text will be again be saved. Of course, not saving altered text is dangerous (but may be required). We therefore get the diagram in Figure 10 with dangerous states hatched.

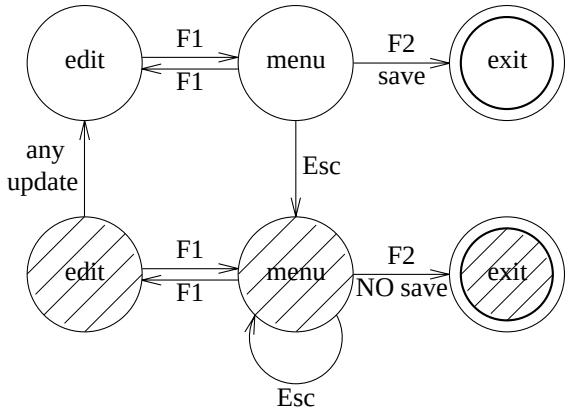

Figure 10: Revised STN with dangerous states

This multiplying of states is a *semantic* distinction, but can be recorded in the dialogue. We can then ask at a dialogue level whether or not it is easy to get into the dangerous states by accident. The user spends most of the time in the edit state, so the most dangerous sequence is 'F1-Esc-F2 – exit with *no* save. This is rather close to the sequence 'F1-F2' – exit *with* save, but is this mistake easy to make?

If we decided it was, we can insert a guard, such as a dialogue box asking for confirmation. In fact, the word processor has no such guard.

The dialogue is *not* as is sometimes claimed independent of presentation. There are various lexical and presentation issues which impinge on the dialogue. In particular, the layout of keys on a keyboard or menu items on a screen affects the sort of lexical errors which occur. For example, the author's old computer had the function keys on a separate keypad. One could not accidentally

hit 'Esc' in the middle of the sequence 'F1-F2'. However, the author's current keyboard layout is as in Figure 11 – disaster!

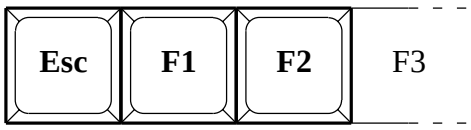

Figure 11: Dangerous function key layout

4 Status/event analysis

We have already seen the state/event dichotomy when discussing dialogue notations. Some are better at states others at events. Really both are needed to understand interactive systems.

Status/event analysis is a semi-formal, "engineering" level, technique which looks at the interplay between status and events in an interface [13, 1]. Status here refers to not just dialogue or system state, but any persistent aspect of the system, such as the display.

Status/event analysis is based on two foundations:

formal modelling – understanding gleaned from variants of the PIE model.

naïve psychology – common sense or very simple perceptual psychology.

It is 'engineering level' in the sense that you do not require a deep understanding of either formal models or psychology.

4.1 Properties of events

Consider alarm clocks. These demonstrate most of the important properties of status and events.

status: the watch face. You can look at the watch face whenever you like.

event: an alarm. It happens at a particular moment, if you aren't there when the alarm goes, you miss it.

status change event: when a particular time passes that is an event, whether or not anyone notices. In general, any change of status can be viewed as an event.

actual and perceived events: you don't necessarily notice an event straight away. There is usually a gap.

polling: if you are waiting for a particular time, you occasionally glance at the watch face. That is you poll the status to make the status change become a perceived event.

granularity: what constitutes an event depends on timescale. If we are interested in planning a day (timescale of hours) then an alarm is an event. However, whilst waking up (timescale of seconds or minutes) the alarm ringing is a status.

4.2 Design implications

Applications have timescales depending on the tasks which they fulfill. Events happen in the system and at the interface. Do these actual events become perceived events for the user, and if so, is the lag within an acceptable timescale for the application?

There are problems if perceived feedback is too slow or too fast.

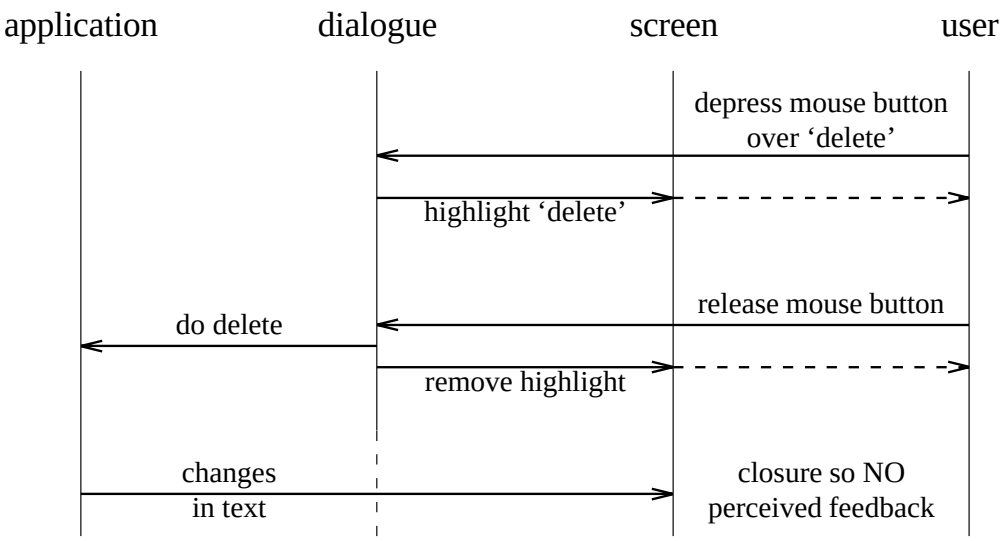

Figure 12: Screen button – hit

too slow – if the lag between actual and perceived event is too great, the recipient may respond too late. For example, if the operator in a power plant were sent an email saying ‘meltdown imminent’, the plant may have vapourised before the operator read the mail.

too fast – if the perceived event is too fast, it may interrupt a more immediate task. For example, if alarms began to ring when the stock of 6mm bolts was running low, then the operator may get distracted from dealing with the potential meltdown.

4.3 Naïve psychology

We can use simple psychology:

- We can predict where the user is looking:

  at the mouse – when positioning

  at the insertion point – intermittently when typing

  at the screen – if you’re lucky

- We know certain effects cause immediate events:

  audible bell – when the user is in the room.

  peripheral vision – movement or large change.

- Closure at the end of a task has predictable effects:

  loss of attention – including the mouse pointer

  concurrent activity – the user may begin a new activity whilst finishing off the last one.

4.4 Example – screen button widget

On-screen buttons invoke an application action. They are activated by clicking the mouse over them However, users often mis-click the button. Furthermore the mistake is often missed. This is a common widget in virtually every graphical interface, and the error is common too. Why?

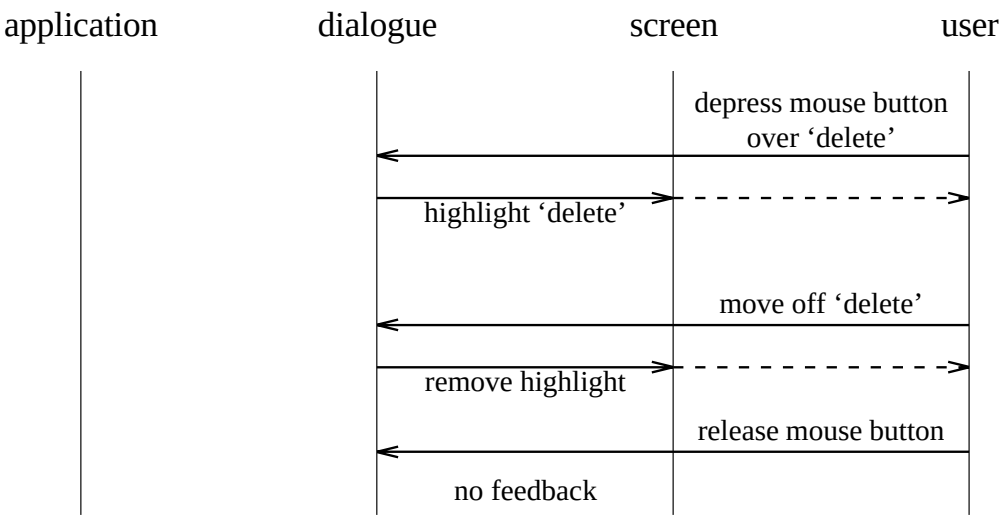

Figure 13: Screen button – miss

To analyse this problem we draw status-event diagrams (Figures 12 and 13). To do this we draw a timeline for each layer of the system: application, dialogue, screen and user. Time runs down the page and arrows between the timelines represent events.

Figure 12 shows the case when the button is successfully activated (a hit) and Figure 13 shows the case when the user slips the mouse off the screen-button just before the mouse-button is released (a miss).

The two diagrams are very similar. They differ in which user action occurs first, the release or the movement of the mouse. However, after the user has positioned the mouse over the target, closure is reached. Therefore the two actions occur concurrently. The mistake is likely.

Why is the mistake not noticed? The highlighting of the button is salient as it is at the target of a mouse positioning task. However, the feedback of application action is not perceived. The user has attained closure and moves on to the next task.

Solution – the button widget must provide feedback when it invokes an application event, for example, a simulated key click. That is the actual event at the application must become a *perceived event* for the user.

Note, this is an expert slip, a novice would explicitly check the application feedback. This means that testing doesn't help to discover or rectify the problem.

4.5 General use of status/event analysis

Status–event diagrams may show different parts of an interface. For example, in an analysis of the arrival of email, the corresponding diagrams had timelines for the file system, the mail program 'mailtool', the screen, and the user. In all cases, status entities (the file system and the screen) mediated the events in the system. Not only was there a gap between the actual events on the screen and the user's perceived event, there was also a gap between the actual event of mail arriving (in the file system) and the perceived event for the mailtool. In general these diagrams are particularly powerful at tracing these gaps.

The application of the status/event distinction is wider than the use of status–event diagrams. Unfortunately, most dialogue notations (and other formal notations) deal primarily with one phenomena or other. However, we have seen that the interactions between status and event phenomena is particularly interesting, emphasising the need for both to be considered together. Status–event diagrams do this for specific scenarios, but one would like also have full dialogue notations dealing with status/event phenomena.

5 Summary

We saw that formal modelling techniques, although powerful and useful, require a high level of formal expertise. In order to 'give away' the benefits of this work to the typical human-factors practitioner less maths' intensive approaches are required.

Dialogue notations of various forms are often used during the interface design process. We have seen how simplified forms of the usability properties can be tested on dialogue descriptions, sometimes with automatic support. Furthermore, the dangerous states example showed how the dialogue description can form a focus for information from both the semantic level (what is dangerous) and the lexical level (what slips are easy to make).

Finally, we saw how status–event analysis can uncover expert slips which are very difficult to uncover during even extensive user testing. Status–event analysis is particularly useful where the interface is not purely reactive. Thus it is especially useful in open-systems and multi-user systems. The author is currently investigating the use of analytic techniques in the area of cooperative working.

Annotated bibliography

General

1. Alan Dix, Janet Finlay, Gregory Abowd, and Russell Beale. *Human-Computer Interaction.* Prentice Hall International UK, Hemel Hempstead, 1993.

   The material in this paper is drawn largely from Chapters 8 and 9 of this book, which expand upon several of the areas.

Formal models of interaction

2. A.J. Dix. *Formal methods for interactive systems.* Academic Press, London, 1991.

   This covers the PIE model and many extensions and other models, including those on which status/event analysis is based.

3. M. D. Harrison and H. W. Thimbleby, editors. *Formal Methods in Human Computer Interaction.* Cambridge University Press, Cambridge, 1990.

   An edited collection covering a range of formal techniques.

4. Harold W. Thimbleby. *User Interface Design.* ACM Press, Addison-Wesley, New York, 1990.

   A wide ranging book which some extensive explicit formal material, and employing a formal approach to problems in much of its informal material.

5. A.J. Dix and C. Runciman. Abstract models of interactive systems. In P. Johnson and S. Cook, editor, *HCI'85: People and Computers I: Designing the Interface*, pages 13–22. Cambridge University Press, Cambridge, 1985.

   The original PIE paper.

6. B. Sufrin. Formal specification of a display editor. *Science of Computer Programming*, 1:157–202, 1982.

   A classic paper describing the formal specification of a display based text editor.

Undo

In case the reader's appetite for the fascinating area of undo has been wetted here are a few papers to read. In addition, see Chapter 2 and 4 of [2] and Chapter 12 of [4].

7. Gregory D. Abowd and Alan J. Dix. Giving undo attention. *Interacting with Computers*, 4(3):317–342, 1992.

   A formal analysis of undo in the context of group editing.

8. James Archer, Jr., Richard Conway, and Fred B. Schneider. User recovery and reversal in interactive systems. *ACM Transactions on Programming Languages*, 6(1):1–19, January 1984.

   A classic paper analysing different forms of undo.

9. Jeffrey Scott Vitter. US&R: A new framework for redoing. *IEEE Software*, 1(4):39–52, October 1984.

   Takes undo and redo to its extreme!

10. Yiya Yang. Undo support models. *International Journal of Man-Machine Studies*, 28(5):457–481, May 1988.

    Informal analysis and review.

Dialogue

As well as the following, see Chapter 8 of [1] which describes dialogue properties in more detail and any book on UIMS.

11. H. Alexander. *Formally-based Tools and Techniques for Human-Computer Dialogues.* Ellis Horwood, Chichester, UK, 1987.

    Describes her SPI notation which is both quite powerful and very easy to read.

12. H. W. Thimbleby. Literate using for finite state machines. University of Stirling, 1993.

    Describes the Hyperdoc tool, which supports simulation, dialogue analysis and automatic documentation.

Status/event analysis

See Chapter 9 of [1] for status–event diagrams and Chapter 10 of [2] for its formal roots.

13. Alan Dix. Beyond the interface. In J. Larson and C. Unger, editors, *Engineering for Human-Computer Interaction*, IFIP Transactions A-18, pages 171–190. North-Holland, 1992. Proceedings of IFIP TC2/WG2.7 Working Conference, Ellivuori, Finland, 10-14 August 1992.

    Relates status/event phenomena to the timescales overwhich they operate and the concept of *pace*.