this chapter appeared as:
'Chapter 2 Formal Methods' in Perspectives on HCI: Diverse Approaches, eds. A. Monk and N. Gilbert. London, Academic Press. 1995. pp. 9-43

Formal methods
an introduction to and overview of the use of formal methods within HCI

Alan J. Dix

abstract

This chapter appeared in Andrew Monk and Nigel Gilbert's collection "Perspectives on HCI". This book contains chapters from experts in a variety of disciplines contributing to HCI. This chapter is about the use of formal methods in HCI.

The chapter is organised around three major uses of formalism within HCI:

- Specification of individual interactive systems
  notations to describe the intended interactive behaviour of specific systems
- Generic models of interactive systems
  models that allow reasoning about broad properties such as the meaning of undo
- Dialogue specification and analysis
  notations, often diagrammatic, for describing the navigational or dialogue structure of applications

The chapter is written assuming a broad audience and the small amount of set and function notation used within it is explained within the chapter.

Table of Contents

this chapter appeared in:
Perspectives on HCI: Diverse Approaches,
eds. A. Monk and N. Gilbert. London, Academic Press. 1995. pp. 9-43

Chapter 2

Formal methods

Alan J. Dix

2.1 Introduction

For many years I have worked on the interplay between formal methods and human–computer interaction. This area of research originated with (present and past) workers from York, but over the last few years there have been several international workshops on the subject and there are now several books on aspects of this area. For further reading in the area the interested reader can consult the chapters on Dialogue and Formal Methods in (Dix et al., 1993), my previous monograph (Dix, 1991) and the collection (Harrison and Thimbleby, 1990).

2.1.1 Why use formal methods?

Formal notations and mathematics are used in several areas of human–computer interaction, including cognitive modelling and task analysis. However, this chapter will focus on those more connected with the engineering and analysis of interactive systems. These notations all try to abstract away from the way the system is programmed, but still be precise about some aspect of its behaviour. Of course, an informal description does the same, but with a formal description you can (in theory) say precisely whether or not a real system satisfies the description. Because of this, one can perform precise analyses on the description itself, knowing that any conclusions one comes to will be true of the real system.

One value of this precision is that it exposes design decisions which otherwise might not be noticed until the system is being implemented. It is clear in many systems that obscure interface behaviour could not have been designed that way, but has occurred as the result of some programming decision. The specification of an interactive system should not determine the algorithms and data structures used to build the system – that is the proper domain of the programmer. But, it should describe precisely the behaviour of the

system – the programmer may not be qualified to make such decisions and the level of commitment at the time that the issue is uncovered may mean that the design choice has been determined by foregoing implementation choices.

2.1.2 Uses of formal methods

We will consider three major strands of formal methods, each of which fulfils a different purpose:

Specification of individual interactive systems

This usually concentrates on a specific system and the complete specification of all aspects of its behaviour. Its purpose is to clarify design decisions, to expose inconsistency and to act as a 'contract' with the implementor. User interface software can be extremely complex and so being able to deal with it at a more abstract level is even more important than for general software.

Generic models of interactive systems

The second strand models classes of system, for example, one might have a general model of window managers as opposed to a specific model of the Macintosh window manager. Their purpose is to give new insight into general problems as the properties of the problem domain are analysed. For example, we will see later how general questions about the meaning of the undo command can be addressed without recourse to a specific system. In addition, they can be used as part of a formal development process to constrain the design of specific systems. That is, results of the analysis of generic models can be applied to formal specifications of specific systems.

Dialogue specification and analysis

Finally, dialogue notations are again used to describe specific systems, but at a different level of detail than a full formal specification. They concern the steps of the user interaction but typically do not fully specify the meaning attached to the user's actions. For example, the dialogue specification of a graphics editor may say that the user must always enter two positions (by mouse clicks) after selecting the 'draw line' icon. However, it will not say that a line appears in the screen, except perhaps by way of informal annotation. Dialogue notations are used for various reasons, but this chapter will emphasise the way the formal element in the dialogue can be analysed in order to expose potential user interface problems.

2.1.3 What is formal anyway?

Of these three strands, dialogue specification is perhaps least mathematical, but most easily used by the non-formalist. Indeed, although formal methods can be extremely powerful they do require a high-level of expertise. Most computing courses now include some element of formal methods and so the level of formal expertise will increase in coming years. However, it is unlikely that there will ever be a large community of people expert in both human factors and formal methods.

This suggests that formal methods need to packaged so that non-experts can get some of the benefits without negotiating the steep learning curve. One way this can be achieved is through 'engineering level' notations which have formal underpinnings, but where simplified analysis and heuristics can be applied. This is as in other disciplines where the practising engineer does not use the theoretical methods and analyses directly, but instead more pragmatic and approximate versions of them. In the user interface domain, dialogue notations are one example of an engineering level notation and are amenable to both simple hand analysis and automated tool support. The fact that many dialogue notations have a graphical form also makes them more palatable! Another example of an engineering level notation is *status/event analysis* which uses simple timeline diagrams together with design heuristics based on a combination of formal analysis and naïve psychology (Dix, 1991, Ch. 10, Dix, 1992, Dix et al., 1993, Ch. 9).

Sometimes the benefits of formal analysis can be presented informally. For example, a purist might argue that an undo button should always undo the effects of the last command — even if the last command was itself undo. However, we shall see later that this is in fact impossible, thus removing the cause of long arguments and allowing more constructive debate over the purpose of undo. Not only can we state the result 'it is impossible' in this case the formal proof can be rendered in a reasonably informal, but convincing manner — which is as well as no-one ever believes the result!

Of course, we can all recognise a bit of formal notation – simply watch out for the λ∀∃! However, you will find that the dialogue notations are mostly diagrammatic. Can a graphical notation be formal? In fact, a diagram can be formal, informal or somewhere in between, depending on the meaning which s attached to the elements of the diagram. This is obvious when we think of an engineering diagram. When it says that the diameter of a rod is 13.7 mm it means precisely that! The dialogue notations discussed in this chapter will be semi-formal in that they have textual annotations on the diagrams which require informal interpretation. However, the structure of the diagrams will be perfectly formal and capable of formal analysis. The

counterside of this also needs to be considered. Just because a paper is filled with Greek and upside down letters doesn't mean it is formal!

Any formal notation abstracts in some way. In being very precise about some things it completely ignores others. The important thing is to be aware of what is being abstracted and whether the abstraction is appropriate for the purpose for which it is required.

In the next chapter a computer game is specified in Z, a particular formal notation. It is thus an example of the first strand of formal methods.[1] In the rest of this chapter we will look at the other two strands in more detail. We will begin with a short introduction to the language and concepts used in formal methods.

2.2 The language of mathematics

The notations used for formal specification in computer science are based on a few key concepts from mathematics. Mathematicians care a lot about the meaning of these concepts, but are typically not worried about the notation used. Indeed, for the same concept, say the application of a function *f* to a value *x* one may see any of the following notations (and probably more besides):

$f(x)$
$f\ x$
$x\ f$

As you see, even the order may change! Furthermore, mathematicians will invent notations for specific purposes, even just for one paper. A piece of mathematics is written for a human reader (well a mathematician anyway) and all that matters is that it is understood by the reader.

Computer science formalists are far more starchy. Different notations exist for the same concepts, but one is normally expected to stick to a particular notation and the proponents of essentially similar notations can become quite tribal at times. The reason for this stickling for notation is that computer science formalisms are written against the background of computer programs where the reader is not another human (or even a mathematician) but a computer. This punctiliousness can be a pain when the notation seems a poor match for the problem but has the advantage that automatic tools can help to check some aspects of a specification.

Unfortunately, several aspects of interface design fit badly with the standard notations and so several specific formalisms have been developed aimed specifically at interface design. The advantage of

[1] In fact, the purpose will be to expose gneral architectural concepts and thus also has aspects of the second strand.

such domain specific notations is that they can have features specifically customised for interface design. However, each new notation requires work in establishing its formal foundations and if required the development of new support tools.

As a mathematician at heart, this chapter will try to not be too heavy on new notation. Where there is a choice of symbol or notation I will use those adopted in the Z notation (Spivy, 1992) which will also be used in the next chapter. However, I will not follow Z slavishly, especially where the Z notation becomes obscure and over-complicated for the examples used here.

Mathematics is like a pyramid stood upon its head: there are a few basic concepts forming the foundation and on these concepts are built successive layers of abstraction. From counting coconuts one moves to the use of numbers, to algebra (talking about numbers in general) to various forms of number-like things and so on. Only the tip of this pyramid is required to understand most formal specifications.

I will assume the reader is familiar with numbers (ordinary numbers!) and with basic logic. I will use the following logical symbols:

$p \wedge q$	–	logical and,	also sometimes written		$p.q$ or $p \cap q$
$p \vee q$	–	logical or,	″	″	$p + q$ or $p \cup q$
$\neg p$	–	logical not,	″	″	$\overline{p}$
$p \Rightarrow q$	–	p implies q	″	″	$p \subset q$

Logic is the glue that joins together mathematics, but the heart upon which mathematics is based are sets and function. Upon these virtually everything else is built.[2] Many readers will be aware of some set theory possibly from school days, so the following descriptions may be familiar.

2.2.1 Sets and other collections

A set is a collection of things which is *unordered* and *without repeats*. The simplest sets are finite ones, for example:

Primary	=	{ *red*, *green*, *blue* }
Secondary	=	{ *blue*, *green*, *cyan*, *red*, *magenta*, *yellow* }
Sea	=	{ *green*, *cyan*, *blue* }

This says that *Primary* is a set with three *elements*, namely: *red*, *green* and *blue*. we say that *red*, *green* and *blue* are *members* of *Primary*. This is written *red* μ *Primary*, *green* μ *Primary*, etc. The second set, *Secondary* is a set with six elements, those in *Primary* and in addition *cyan*,

[2]In fact, sets are sufficient on their own, but why save on shoe leather by walking on your hands?

magenta and *yellow*. The third set, *Sea* consists of the palette one might want for a sea-scape. Note that as sets are unordered we could have written the elements in any way and had the same set:

Primary = { *red*, *green*, *blue* } = { *green*, *red*, *blue* }
= { *blue*, *green*, *red* } = …

In addition, as the set does not count repeats we can only have one of each colour. If we wrote:

BlueSet = { *blue*, *red*, *blue*, *green*, *blue* }

this would be regarded as exactly the same as *Primary*, the repeated *blue*s are ignored. Alternatively, the definition of *BlueSet* might be regarded as ill-formed.

As *Secondary* contains all the elements that *Primary* does, we say that *Secondary* is a *superset* of *Primary*, or equivalently that *Primary* is a *subset* of *Secondary*. This is written:

$Secondary \supset Primary$ or equivalently $Primary \subset Secondary$

In addition, note that $Sea \subset Secondary$.

We can build new sets from existing ones. This is done using set *union* (written ⋃) which puts two sets together and *intersection* (written ⋂) which extracts those elements common to two sets. For example:

$Primary \bigcup Sea$ = { *red*, *blue*, *green*, *cyan* }
$Primary \bigcap Sea$ = { *blue*, *green* }
$Sea \bigcap Secondary$ = { *blue*, *green*, *cyan* }

Note that because *Sea* is a subset of *Secondary* the intersection of *Sea* and *Secondary* is the same as *Sea*.

Not all sets are finite, for example the set of all *natural* numbers (non-negative integers) or the set of all integers:

$\mathbb{N}$ = { 0, 1, 2, 3, 4, 5, … }
$\mathbb{I}$ = { …, -3, -2, -1, 0, 1, 2, … }

Usually infinite sets represent something abstract, after all it is difficult to really gather together an infinite collection of things. In fact, as infinite sets go, the set of natural numbers is relatively small and concrete.

In addition to sets it is sometimes useful to deal with *bags* which are unordered, but do allow repeats and *sequences* which are both ordered and have repeats.[3] So, one can write:

BlueBag = œ *blue*, *red*, *blue*, *green*, *blue* “
= œ *red*, *blue*, *blue*, *blue*, *green* “
= œ *green*, *blue*, *blue*, *red*, *blue* “

You can think of a bag as precisely that, a sack containing things in no particular order, but more of one thing than another. So *BlueBag* contains three *blue*s, one *red* and one *green*.

One can have a similar blue sequence:

BlueSeq = ⟨ *blue*, *red*, *blue*, *green*, *blue* ⟩

However, this time the order is important. A sequence is like a list on a piece of paper. The first item on the list is ‘blue’, the second is ‘red’, the third is ‘blue’ again. So, if we swop the first two colours we get a different sequence:

BlueSeq ≠ ⟨ *red*, *blue*, *blue*, *green*, *blue* ⟩

Similar to a sequence is a tuple. It too is an ordered collection. However, the elements of sequences (and bags) all the same type of thing, for example, *BlueSeq* consists solely of colours. A tuple may contain different sorts of things. For example, if we wanted to describe the colour an size of a dress, we might do so using a tuple:

dress = (red, 10)

When specifying systems, tuples will be used to describe the state of a system in terms of its component parts. Often the components of a tuple are given names, for example, we may want to talk about *dress.colour* and *dress.size*. The notation for describing named tuples varies more than most although the use of a dot to access components is quite widespread.

I said that mathematics was like an inverted pyramid. We start with sets of things and other sorts of collections. However, from these one then goes on to talk about sets of sets, sets of sets of sets, …, not to mention sets of sequences, sequences of sets etc.

Lets start with the set $X = \{ x, y \}$. From this we can obtain $\prod X$, the *power set* of X — the set of all subsets of X , and **seq**X — the set of all finite sequences of elements of X .

---

[3] You could also have a collection which is ordered but has no repeats, a sort of prioritized set, Z calls this an injective sequence (**iseq**), but I have never found a use for one!

$$\prod X \;\&= \&\; \{\; \{\}\;,\; \{\; x\;\},\; \{\; y\;\},\; \{\; x, y\;\}\;\}$$
$$\textbf{seq}\; X \;\&= \&\; \{\; \langle\rangle\;,\; \langle\; x\;\rangle, \langle\; y\;\rangle, \langle\; x, x\;\rangle, \langle\; x, y\;\rangle, \langle\; y, x\;\rangle, \langle\; y, y\;\rangle, \langle\; x, x, x\;\rangle, \ldots\;\}$$

Note that the power set of a finite set is also finite, but that the set of all sequences is infinite. However, even with finite sets power sets grow very rapidly in size and hence are used to describe problems but are rarely actually computed within a program. Note also that the empty set {} (also written ∅) is the set which contains nothing. It is a subset of every set including itself.

We can also construct the set of all tuples using *Cartesian product*. So, if $A = \{\; a,\; b\;\}$, the Cartesian product of A and X is the set of all tuples whose first component is from A and whose second component is from X: $A \times X = \{\; (a,x),\; (a,y),\; (b,x),\; (b,y)\;\}$. Note especially that this does *not* contain (x,a) — which is a member of $X \times A$. The set of all dress styles could then be written as $Styles = Primary \times \bar{}$ (recall that $\bar{}$ was the set of all natural numbers). Of course, one could argue with this as we may want to have dresses in non-primary colours. Also some of the sizes in this set do not represent reasonable dresses (have you ever seen a size 593 dress?). This is precisely the value of a formal specification. Often this sort of decision is taken while a system is being programmed rather than by the designer.

When a name is introduced in a specification it is usually given an explicit type to say what sort of thing it refers to. For example, we might introduce the name *sty* which is going to be a specific dress style. This can be declared as:

$$sty : Primary \times \bar{}$$

This says that *sty* is some tuple consisting of a primary colour and a natural number. A possible value of *sty* is then (*green*,8). Note the difference between *sty* and *Styles* (defined above) — *Styles* is the *set* of all possible tuples whereas *sty* is one particular tuple. In fact we could have used *Styles* as the type of *sty*:

$$sty : Styles.$$

2.2.2 Functions and relations

Functions are black boxes which given an element from one set return an element of another. Most readers will have come across some mathematical functions, for example, *square* the function which given a real number (that is any number not just integers) returns its square, e.g., *square*(1.5) = 2.25. The type of a function is shown using an arrow and the function, for example the type of *square* might be $\Omega \rightarrow \Omega$, that is it takes real numbers (Ω) to real numbers.

Often functions take several parameters and this is denoted using the Cartesian product notation × which we saw earlier. For example:

$$mix : Primary \times Primary \rightarrow Secondary$$

says that *mix* is a function which given two primary colours returns a secondary colour. For example, if *mix* is standard colour mixing (for light rather than paint) we would have *mix*(*red*,*green*) = *yellow*.

If the sets involved are finite we can write out the function in full. For example, *mix* can be defined in full by:

$$\begin{aligned} mix = \{\ & (red,red) \mapsto red, & (red,blue) \mapsto magenta, & (red,green) \mapsto yellow, \\ & (blue,red) \mapsto magenta, & (blue,blue) \mapsto blue, & (blue,green) \mapsto cyan, \\ & (green,red) \mapsto yellow, & (green,blue) \mapsto cyan, & (green,green) \mapsto green \quad \} \end{aligned}$$

Note that it is necessary to specify both *mix(red,blue)* and *mix(blue,red)* as in general these need not be the same. For example, subtraction (–) is a function of type $\Omega \times \Omega \rightarrow \Omega$, but 3–2 is not the same as 2–3.

In fact, expressing functions in full is the exception usually they are described using some formula. For example, the function *hypot* might return the length of the hypotenuse of a right-angled triangle given the length of its two sides using Pythagoras' formula:

$$hypot: \Omega \times \Omega \rightarrow \Omega$$
$$hypot(a,b) = \sqrt{a^2 + b^2}$$

Notice that the first line gives the type of *hypot* and the second line its definition. Note also that *a* and *b* represent variables in the above definition whereas in a statement like *mix(blue*,*green) = cyan*, the colour names were constants. Specific notations have rules for distinguishing these, but for this chapter I hope it will always be clear from context.

Functions may not be defined for all values in which case they are said to be *partial* as opposed to a *total* function which is defined for all values of the given type. However, a function must always give the same answer. So if we considered the set of all people *mother* would be a reasonable function as everyone (new IVF developments not withstanding) has a mother. However, *daughter* would not be a function as not only does not everyone have a daughter (which would simply make the function partial), but also some people have more than one daughter so there is not a single result.

Obviously one wants to deal with relationships like *daughter* and there are two effectively equivalent constructs for this. The first, in the spirit of inverted pyramids, is to have a function *daughters* which

returns a *set*, so that the result of *daughter* is the set of daughters of a person. If the person has no daughters the set is empty. For example,

daughters(Alan) = { *Esther*, *Ruth* }
daughters(Janet) = {}

That is Alan has two daughters, Esther and Ruth, Janet has none.

The other way to deal with the problem is using a *relation*. We could have the relation *daughter_of(c,p)* which is true precisely when *c* is the daughter of *p*. For example, using the same family relations as above, both *daughter_of(Esther,Alan)* and *daughter_of(Ruth,Alan)* would be true, but *daughter_of(Janet,Alan)*, *daughter_of(Ruth,Janet)* and *daughter_of(Alan,Esther)* would all be false. The type of a relation is sometimes written using a double ended arrow:

daughter_of : *Person* ¢ *Person*

We have already seen a function yielding a set, that is daughters. Its type can be written using the power set construction:

daughters : *Person* › ∏ *Person*

and yes, you guessed it, you can have sets of functions, functions returning functions, sets of functions returning sets, ad infinitum!

2.2.3 States and operations

The most common use of formalism within computer science is to define the possible states of a system and the operations which can change that state. In other areas of computer science the part of the state which involves the user interface is often ignored. For interface design and analysis, we want to talk about everything that concerns the interaction.

As an example, consider what you would need in the state of a simple calculator. This of course depends very much on the particular calculator, but would at least include: the number displayed, the current running total, the operation about to be performed and something to say whether the next digit will be added to the end of the current number or replace it. Did you think of the last two? The operation pending is needed because after typing ‘1’, ‘+’, ‘2’ the calculator needs to remember that it must add 1 to 2 if you type ‘=’ next. The last part, the ‘typing’ flag is needed to tell whether typing a ‘7’ next will lead to ‘27’ being displayed (to be added to 1) or whether the ‘2’ will be replaced by ‘7’.

The state is changed by each user action. A specification will say precisely how each part of the state is changed by each operation. For example, the rule for typing a digit would be something like:

type_digit(d) **if** typing flag is true
then add d to the end of the number displayed
if it false
then clear the number displayed and set it to d

A typical trace of operations might proceed as follows:

user action	running total	display	operation	typing?
< *start* >	1	2	+	yes
type_digit(7)	1	27	+	yes
=	28	28	none	no
–	28	28	–	no
type_digit(3)	28	3	none	yes

Notice how the effect of typing the '7' and the '3' are different because of the typing flag.

The state of the calculator is a tuple, for example, the state just after typing the '7' is:

(0, 28, +, yes)

The set of all states is therefore a Cartesian product:

$$CalculatorStates = \mathbb{1} \times \mathbb{1} \times \text{Operation} \times Flag$$

and an operation can be regarded as a function from states to states:

$$\text{type\_digit}(7) : CalculatorStates \rightarrow CalculatorStates$$

This function says how the states changes when the operation is performed.

Often some parts of the state can be thought of as the underlying application whereas other parts are to do with the process of interaction. For example, the running total is obviously what one really wants out of the calculator, whereas the typing flag is part of the ephemera of interaction. Often the bits describing the interaction are the most complicated – in order to make something that is simple for the user, the system has to be complex.

When the system gets more complicated we don't want to have to write the full state in one go. For example, a CAD package may have the following state:

$$CADstates = MenuState \times Selection \times Drawing \times Notes$$

The components may be named, so if *cstate* is a particular state of the CAD system, we may be able to refer to *cstate.menu*, *cstate.sel*,

cstate.draw and *cstate.note*. The various sets *MenuState* etc. refer to possibly complex sets representing parts of the state. For example, *Selection* would have information regarding the currently selected object in the drawing, and *Drawing* would record all the shapes and lines which have been drawn. These would each be complicated sets in their own right. A full specification notation makes it easy to define these sets one by one and hence build up the whole specification. In the next chapter, Z's structuring mechanism *schemas* are used to build a complete specification.

In this example too we can see that some parts of the state, *cstate.draw* and *cstate.note* correspond to the underlying application and the rest *cstate.menu* and *cstate.sel* are the state of the interaction.

Note however, that this distinction is an area where authors of HCI papers using formalism often appear confused. They begin to talk (correctly) about the changes that happen to the system in terms of transitions between states. However, at some point they want to refer to the interaction component, but do so by talking of it as a *subset* of the states. That is, the states are classified into interaction states and application states. They basically say something like the following:

$$\begin{aligned} Istates &\subset CADstates \\ Astates &\subset CADstates \end{aligned}$$

This is entirely wrong — it is the components of the states which can be classified. Think what they are saying, the set of states represents possible snapshots of the system at any moment in time. To classify the states would be to say that at some moment the system had only interaction state or only application state. The latter is bad enough, but if the former were true what would have become of all your lovely drawing! Certainly the human-computer dialogue at any moment may be concentrating more on surface interaction (e.g., menu manipulation) or deep interaction (e.g., invoking a structural analysis), but the other components of the state are also present even if unchanged.

Unfortunately, this is not simply a matter of the authors knowing what they want but having trouble translating it accurately into the formal notation. Personally, that wouldn't bother me too much so long as the intent is clear — and to be honest I've seen far worse in papers on software engineering where formal methods is supposed to be the central focus! However, there are occasionally points where the errors represent a deeper confusion, not just about the mathematics, but in the authors' general thinking. I have several times seen pictures similar to Figure 2.1, which exactly captures this idea that some of the states (the black ones) are solely to do with the underlying application and others (the white ones) are to do solely with interaction.

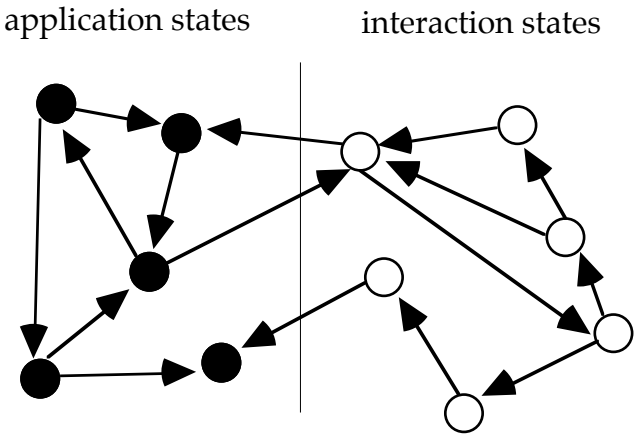

Figure 2.1 The WRONG way of thinking about state!

The confusion arises because at any moment of time one may be able to divide up the state of the system into an interaction part and an application part, which is in a sense looking at a 'subset' of the components. But this process is selection of components and is a process of abstraction. If one wants to talk generically about the relationship between the state of the system and the 'interaction state' one needs an abstraction function to obtain the relevant portion.

$$iabs: CADstate \rightarrow MenuState \times Selection$$

This difference between abstraction (only looking at some aspects of things) and selection (only looking a some things) can be subtle, but is important in both our formal and informal thinking.

2.2.4 Specialist notations

So far we have been talking about standard mathematics and the specification in the next chapter will be in Z a general purpose formalisms. However, there are also several special purpose notations which can be useful for interface design. Notice the specifications only considered either single states of the system or simple transitions between states. It is slightly more complex to talk about sequences of actions — not impossible, for example, one can use sequences of states to represent the history of an interaction, but certainly more difficult. The dialogue notations discussed at the end of this chapter, are aimed at the specification of the possible sequences of user actions. However, they specify precisely what a system does, but are not suited to describing *constraints* on the temporal behaviour.

Various forms of temporal and modal logics are better suited to this and allow one to make statements like: "it is always true that when

the user presses the ‘print’ button the document will eventually be printed”. In temporal logic one can write this as:

$$\Box\ \text{user presses ' print'} \Rightarrow \Diamond\ \text{document is printed}$$

The symbols ❑ and ◊ are read `always' and `eventually' respectively. Note that `eventually' means precisely what it says, not instantly, not even necessarily today, but … eventually. However, there are stronger statements that can be made! Johnson (1992) has made extensive use of temporal logic in specifying and prototyping safety critical process control interfaces.

Special purpose interface notations usually involve some sort of notation to talk about sequence of actions, but more often of the dialogue notation rather than the temporal logic style. In addition, such notations may allow the interface to be defined in terms of semi-autonomous agents (Abowd, 1990, 1991). This corresponds more closely to the object-oriented style of many interfaces, but has the disadvantage of being more program-like.

I have been particularly involved in two areas where current notations seem particularly weak. The first is in expressing properties such as the dragging of an icon across a screen with a mouse. In the style of specification used above, this would have to expressed as lots of little mouse movement events which each change the state by a little, and hence change the display. This description using small scale events does not adequately reflect the fluid feel that such an action has for the user. This is an example of a status-status mapping within the interface, which reflects the variation of certain phenomena *between* events. Effective notations to describe both event and status phenomena is a current area of research (Dix, 1992).

The second area is in the description of asynchronous groupware. The multi-user spreadsheet was an example of synchronous groupware as the effects of one user's operations were assumed to be instantly visible (if they happen to be on screen) to other users. However, when network delays are considered there are always small discrepancies between the state of the system at different users' machines. In addition, some cooperating users may have machines which are not permanently connected at all, communicating by email or floppy disk transfer. Extensions to temporal logic are being formulated to deal with such distributed applications (Dix 1994).

2.3 Generic models of interaction

We found ourselves in the last section beginning to discuss formal properties of multi-user systems at quite an abstract level. We moved

from the discussion of shared spreadsheets to formulations of properties which could apply to any system. The models we will look at in this section are designed specifically for that purpose.

2.3.1 The PIE model

The simplest such model is the PIE model which was developed at York nearly ten years ago (Dix and Runciman, 1985). The letters used to denote various parts of the model are retained for historical reasons (if I changed them the model would no longer be a PIE). However, I won't bother to quote all the full names which go with the acronym as they now serve only to confuse.

The PIE model is based on a black-box model of an interactive system. That is it does not look at the internal workings or structure of the computer, merely at what goes in (user inputs) and what comes out (the display and other outputs such as printed documents) (see Figure 2.2).

Whereas the specification of the spreadsheet was constructed out of various components, the PIE model does not look inside the systems state and simply demands that there is some set *E* of states. Similarly, it does not say what the display is like merely that there is some set *D* of legal displays. The function giving the current display from the current state is called *display* (happily not all the names are obscure). The display is intended to cover any form of immediate output whether visual or aural.

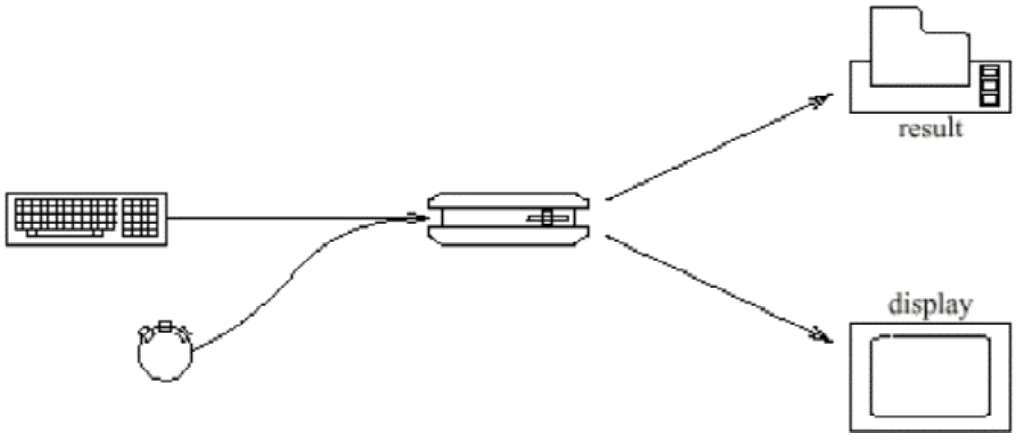

Figure 2.2 Inputs and outputs of single user system

The model distinguishes the display from the results of the interaction, these are the outputs which persist beyond the interaction, for example, the printed form of a document. However, this is in a sense potential rather than actual, as there is always assumed to be a result that you would get if you stopped the interaction now. The set of such potential results is called *R* and is obtained from the current state *E* by the function *result*.

These functions can be given mathematical types:

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

The users inputs are called commands. The model can be used at various levels of granularity, for example, the commands may be individual keystrokes, or may be similar to the operations on the spreadsheet. In the case of a spreadsheet the set of commands *C* would be something like:

C = { *move_left, move_right, …, insert_col,*
set_formula(`'A1+B2'`*), set_formula(*`'57'`*), …* }

Notice that operations like *set_formula* which required extra parameters are transformed so that each instance is a separate command. At the keystroke level the set of commands might be:

C = { `'a'`, `'b'`, …, `'0'`, `'1'`, …, *Ctrl_A, Ctrl_B, …* }

The current state of the system is a function of the history of all commands which have ever been entered. This command history is called *P* and the function is called *I*.

P = **seq** *C*
I: $P \rightarrow E$

This gives all the elements of the PIE model, depicted graphically in Figure 2.3, … which looks rather like the original illustration.

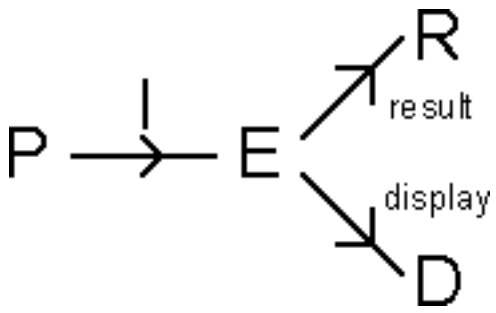

Figure 2.3 The PIE model

One does not normally think of the state as being generated from the history of user operations and specifications are not usually written this way. Instead, one considers the state transition which results from individual operations. The two views of interaction are useful in describing different properties, and so we also define the state transition function *doit* which says how a sequence of commands alters the state:

$$doit: E \times \mathbf{seq}\ C \rightarrow E$$

Given the transition function for individual commands one can easily generate the transitions for longer sequences. For example, if *a* and *b* are two commands, we can work out the effect of ⟨*a*, *b*⟩ on a state *s* by:

$$doit(s, \langle a, b \rangle) = doit(doit(s, a), b)$$

The PIE model has been used to analyse a range of properties. One of the first was the meaning of WYSIWYG — "what you see is what you get", and the structure of the model was to some extent oriented towards this, representing the display — what you see, and the result — what you get. Several variants formalising aspects of this have been produced and details can be found in (Dix 1991). The basic flavour of these properties is that there must be some relationship allowing you to infer the current result from the display. The simplest is to demand the existence of a function *predict* such that:

$$predict: D \rightarrow R$$
$$\forall s \in E \bullet predict(display(s)) = result(s)$$

Let's consider what this says in detail. The first line says that *predict* is a function which from a display will give you a result. The second line uses Å (the universal quantifier), which is read as 'forall'. It says that if you consider any state *s*, take the display of that state and then apply the *predict* function to that display, then the result you get is exactly the same as if you had applied the *result* function directly.

In a very simple drawing package, a tracing of the screen would be exactly the same as the printed drawing. The process of drawing is effectively the *predict* function – you can predict the exact form of the printed output from the display.

This particular formulation is rather strong as it says that you can *completely* determine the result from the current display. This would not allow the printing of off-screen information. For example, imagine the drawing package had a larger drawing than would fit on a screen, with scrollbars to move about it. At any moment, the display would only show a part of the drawing and hence one could not predict all of the printed form from the *current* display. In reality, one would simply scroll back and forth over the picture in order to see it all. Variants of this predictability property can capture this sort of behaviour, but are slightly more complex.

Examining the limitations of formal properties, as we did above, not only reveals problems with the formulation of the mathematics, but also exposes real usability issues. For example, one of the early

attempts to extend the predictability property asked whether it was possible to predict the result from *all* the possible displays which had that result. This was meant to mimic the user's ability to scroll over all of a document or drawing. However, an attempt to prove this over very simple systems exposed a class of interface problems. In general, you not only need to know what displays are possible, but which part of the document they referred to. This problem we called *aliasing* – you can't tell the identity of something from its content – and it arises in a wide variety of contexts. For example, many early text editors did not have a status line giving the current location. So, in a large file of numerical data it was very easy to get lost. Although most modern editors have scrollbars which show the location, these are severely limited for very large documents. There are typically only a few hundred pixels on the scrollbar, so, if the document has more lines than this, the scrollbar only gives a general idea of where in the document you are. Similar problems arise when, for example, two copies of a document are made – if the name of document is not very salient it is easy to change the wrong one!

Another set of properties concern *reachability*, that is what you can do with the system and how hard (or impossible) it is to get from one state to another. Undo is a special case of this as it concerns how one gets to the previous state in an interaction.

2.3.2 Undo

It is widely agreed that user interfaces ought to include some sort of undo facility. This not only allows recovery from errors, but gives the user confidence to explore new parts of the system. There have been several models of undo including various forms of redo (Archer et al., 1984, Vitter, 1984, Yang, 1988}. However, we will only look at fairly simple undo.

The basic requirement for an undo command is that it reverses the effect of the previous command. In other words, if *c* is a command, then the sequence *c* followed by *undo* has no effect. This can easily be formalised in the PIE model:

$$\forall s \in E \cdot doit(s, \langle c, undo \rangle) = s$$

That is whatever state, *s*, you start in performing *c* and then *undo* leaves you in the original state.

Of course, for undo to be useful it must work on all commands. Thimbleby (1990) tells the sad story of a paint program. It had an undo button which worked for all simple cases (where it wasn't really necessary). But, when one day he accidentally performed an area fill which wiped out the whole picture, he found that the undo button was disabled.

This suggests that the above formula should be true of *any* command at all. This at once raises the issue of whether the *undo* command itself is included — that is, is undo undoable? The purist would argue that for consistencies sake all commands should be equal. Indeed, if one accidentally hit the undo button surely one would like to be able to undo it. Furthermore, the undo button on many systems appears to function in this way, if you hit it a second time it restores the system to the state before the first undo. Or does it? Let's look a little more closely.

Assume our system does indeed obey the undo property for all commands *including* undo itself. Consider an arbitrary start state, call it s_0 and any two commands, say *a* and *b*. If we issue *a* from state s_0 we would get to some new state of the system, call it s_a whereas if we had issued *b* we would have got to a state s_b. However, from either state the *undo* command should return us to state s_0. This situation is shown in Figure 2.4. Notice that whichever way you go round the lozenge you get back to s_0. The top and bottom path around it represent different possible traces of user behaviour. But, what now happens if the user enters a second *undo*? Looking at the top of the lozenge, corresponding to the trace where the user entered *a*, the rule that *undo* followed by *undo* has no effect would suggest that the resulting state (denoted by "?") is s_a. However, if we look at the bottom of the lozenge, we would conclude that the state is s_b. So if our system satisfies the undo property states s_a and s_b must be identical. However, the commands *a* and *b* were not special, so whatever commands the user enters in state s_0 it always ends up in the same state (call it s_1). Of course, s_0 was also an arbitrary start state, so the system always does the same thing no matter what command the user enters. Given that one of those commands, namely *undo*, always returns one to the previous state, this implies that the system is either a flip-flop, with only two states (rather boring) or has only one state (i.e., it does nothing at all — even more boring).

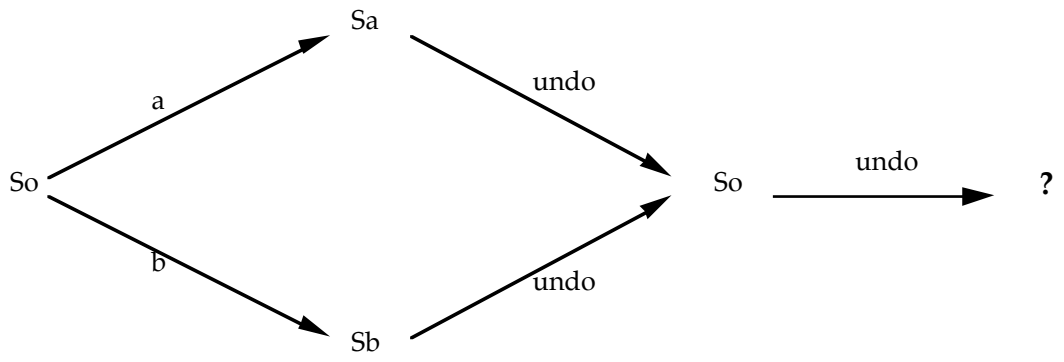

Figure 2.4 Undoing undo

Perhaps you feel you've been hoodwinked by the formal argument. Surely the above is just a limitation in the formalisation of the problem. The diagram represented two possible ways the system could be used, of course in reality only one could happen. Let's say it was the top path, the user does *a* followed by two *undo*s. Now when the user performs the second *undo*, surely you might say, the system knows the user did the action *a* and can thus redo it? But, if this were the case, then the system would have to explicitly remember that *a* had been done – and this knowledge would have to be in the state – just as your memories are in the internal 'state' of your brain. However, if the system were remembering the *a* after the first *undo*, it could not be in the same state as it was at the start (s_0). So, either the first *undo* doesn't forget the *a* properly, or the second one can't remember it in order to redo it! The formal argument is really just a more precise and conclusive version of this argument. The nice thing about the formal argument is that, once you have overcome the hurdle of the formalism itself, it is particularly easy to discuss different potential sequences of actions which get quite involved when expressed textually.

So, in short, we can conclude that *no* reasonable system can *ever* have an undo command that works uniformly for all commands including itself. This is an extremely general and powerful result. It means that one can stop trying to endlessly fix and refix algorithms which attempt to obtain this impossible goal. Instead, one can concentrate ones efforts on developing forms of undo which are both achievable and useful.

If this is so, then what about those systems which appear to have undoable undo? In fact, if you look carefully 'appear' is the right word. The way they work is more or less as follows. Imagine the system without an undo command. It has some system state associated with it. When the undo command is added to the system the state is made more complex and effectively contains *two* (different) copies of the original state. These correspond to the current state and the last state. All the undo command does is to swop these two copies. However, the effect of a normal command is effectively to push out the old previous state and make a new current state. The undo command does not reverse this process.

Notice that in this explanation there is a difference between the state of the system that you normally think of, and the full state of the system when we take the undo command into account. The undo command reverses the effects on the former, not the latter. Like version control and other history or auditing mechanisms undo is best thought of as a meta-command that operates on a completely different level.

As well as being able to study single-user undo it is also possible to look at the meaning of undo in a multi-user context. One problem is the meaning of undo when the user (say Alison) who issues the undo command is not the same user (say Brian) who issued the last command. Should Alison's undo operate on Brian's command — called *global undo*, or on her own last command — called *local undo*. In most circumstances it is clearly local undo which the users will expect. However, there may be some form of interference between commands, so it is not clear when local undo is meaningful. A formal analysis of multi-user undo (Abowd and Dix, 1992) has exposed precisely those circumstances when local undo is possible and also shown that there are circumstances when there is no sensible meaning to it. This impasse acted as a spur to look more closely at what undo is *for*, and hence a complete re-evaluation at an informal as well as a formal level of what facilities ought to be offered for undo.

The example of undo shows how useful formal models can be as tools for understanding. The specification we originally gave for undo 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.

2.4 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.

I said earlier that all formalisms abstract away some details in order to emphasise and make precise others. Dialogue notations abstract away most of the details of the system state in order to emphasise the user's actions on the system and the order in which they can occur.

Even within dialogue notations there is some variation of level of abstraction. Some deal with low level user events such as mouse clicks and keystrokes, others deal with more abstract events such as 'enter login name'. Also some dialogue notations do capture some aspects of the state, especially those which are intended for prototyping. However, in the latter notations the state description is usually clearly separable from the 'real' dialogue description. More normally the effects of user's actions on the system state is denoted by textual annotations or by the use of meaningful names.

2.4.1 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:

circles – denoting the states of the dialogue
arcs – between the circles, denoting the user actions/events

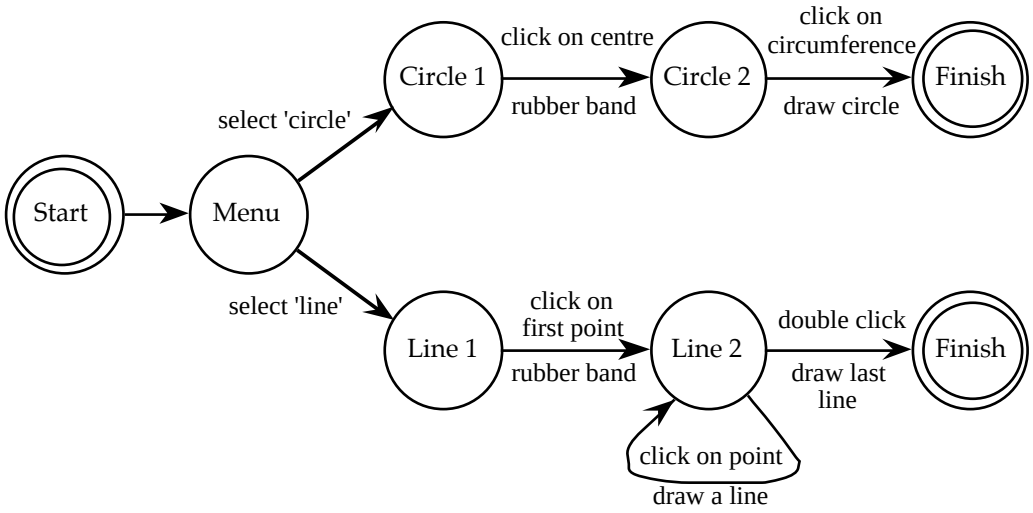

Figure 2.5 State transition network for menu driven drawing tool

Figure 2.5 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 2.5 shows the higher level dialogue for the drawing tool, selecting between several sub-menus. The menu in Figure 2.5 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.

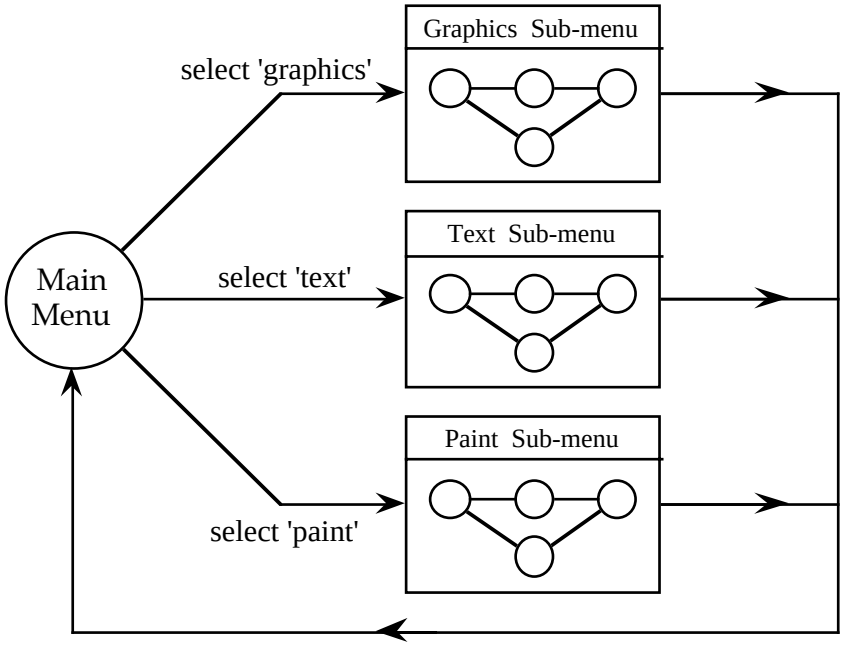

Figure 2.6 Hierarchical state transition network for complete drawing tool

2.4.2 Why do people use dialogue notations?

One advantage of performing formal analysis on dialogue descriptions is that they often 'come for free', a natural product of the design process. There are several reasons for this which we'll look at 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.

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 2.6 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.

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.

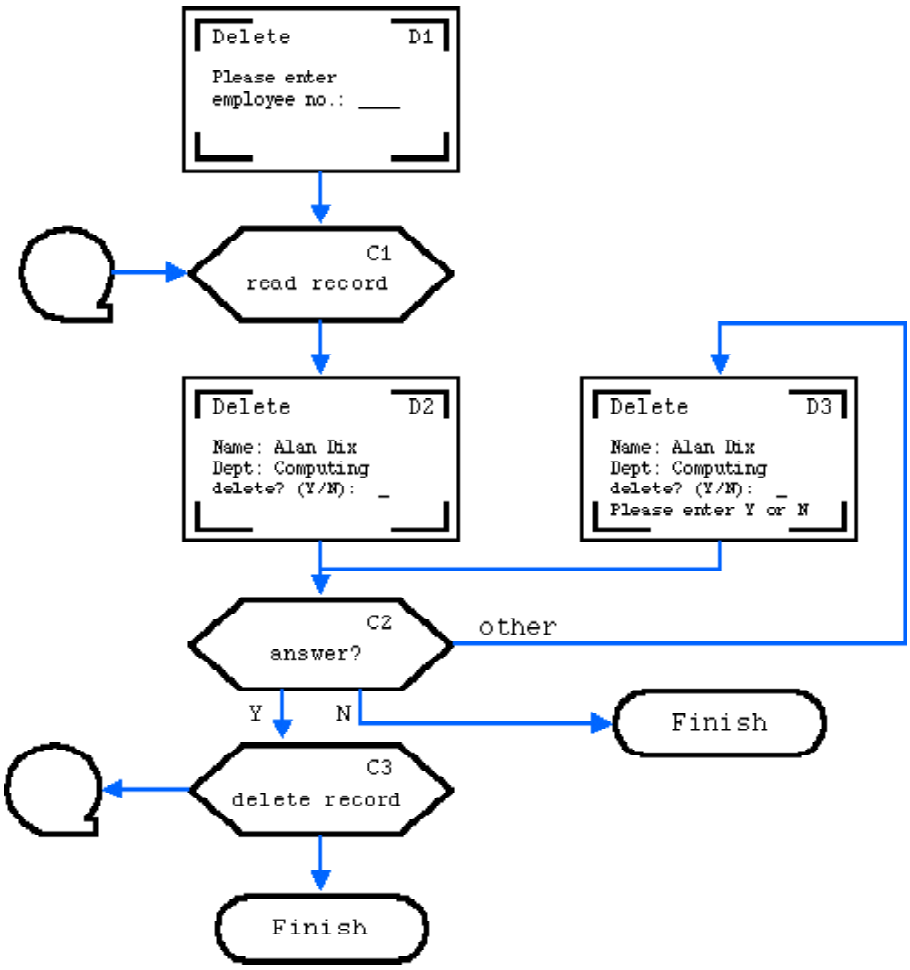

Figure 2.7 Flow chart of deletion sub-dialogue

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) (Alexander, 1987). 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 (Thimbleby, 1993), shown in Figure 2.8. 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'.

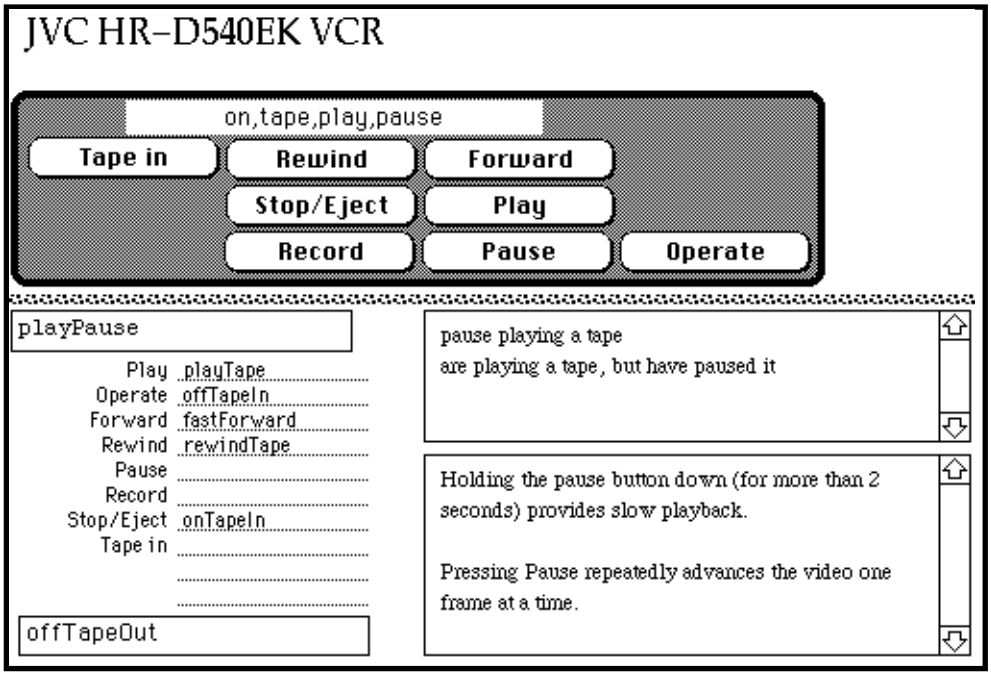

Figure 2.8 Hyperdoc

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

2.4.3 Dialogue properties

Given a dialogue description, we can begin to look at what properties it satisfies. There are several dialogue properties which are to do with local dialogue actions from a single state:

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 2.4, 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 signalled to the user as an error by a

beep, simply ignored, does it do something odd (a feature!) or does it crash the program?

Another set of properties are more global, considering how easy or difficult it is to get from one state to another, and often encompassing whole trains of actions.

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 2.5 and 2.6). 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.

2.4.4 Example – digital watch

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

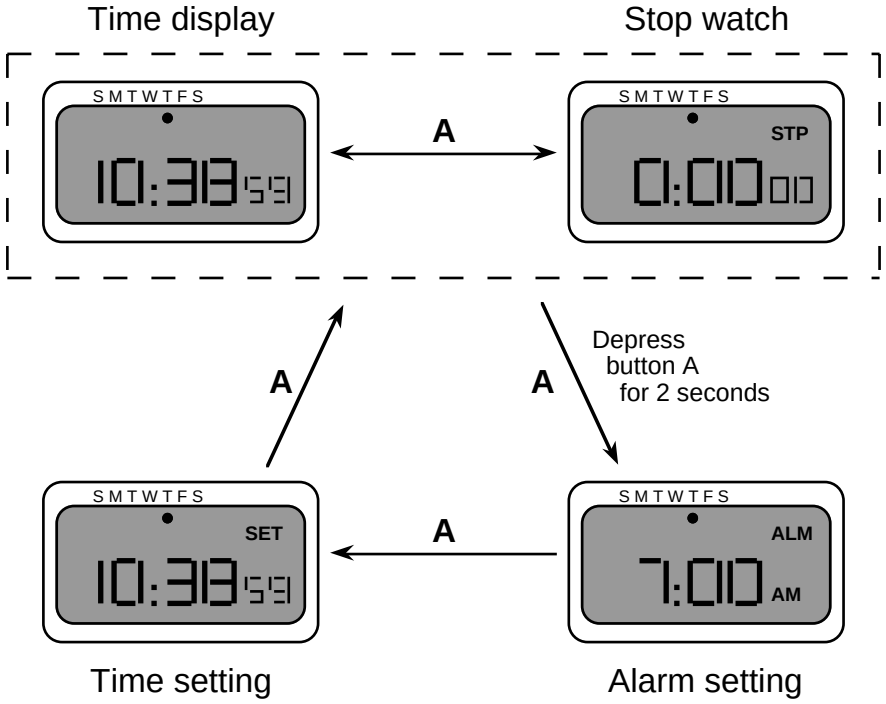

Figure 2.9 Instructions for digital watch

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.

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.

Designer's documentation

Extensive experimentation eventually revealed the complete STN for the watch, shown in Figure 2.10. This includes for each state the effect of the three actions:

- depress A
- release A
- wait two seconds

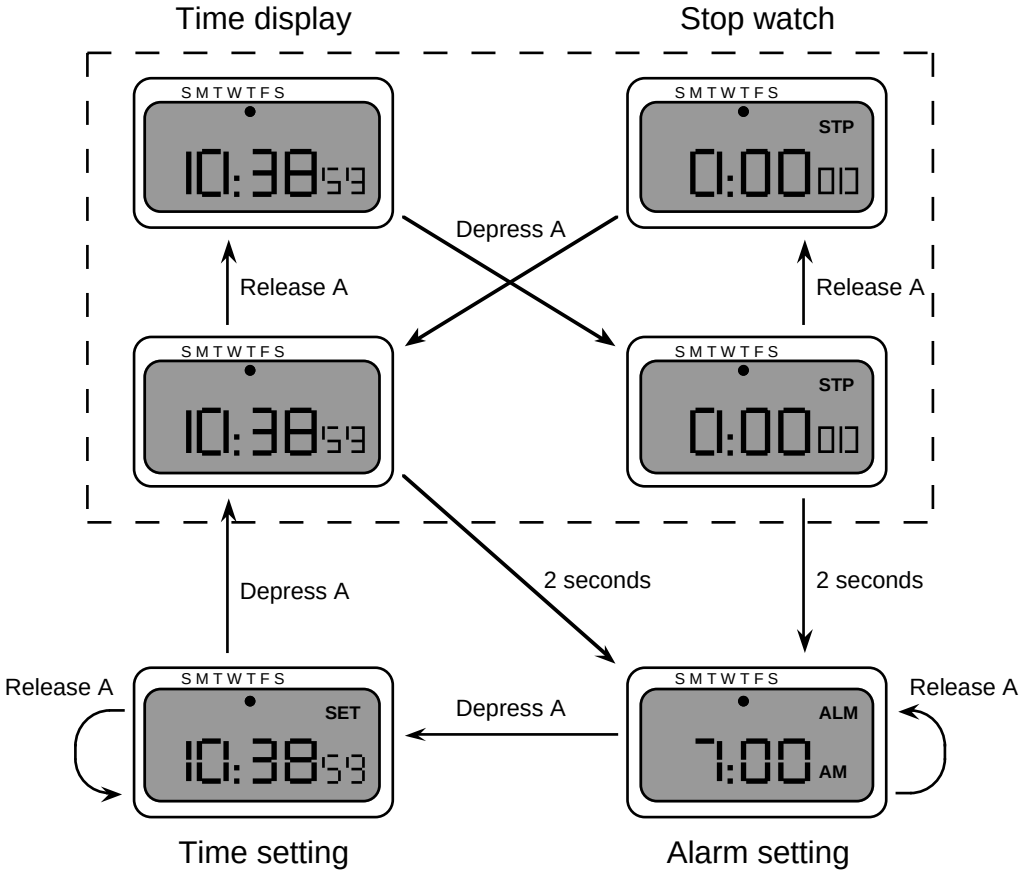

Figure 2.10 Design diagram for digital watch

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!

2.4.5 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 2.11.

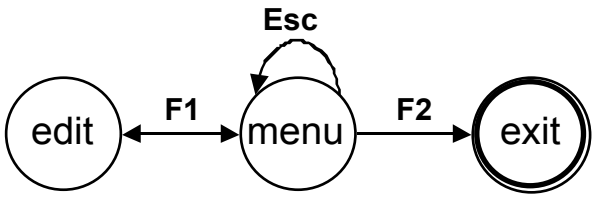

Figure 2.11 Main modes of text editor

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). In order to expose this behaviour the diagram must be redrawn with the states duplicated to differentiate exit with and without save . We therefore get the diagram in Figure 2.12, in which the dangerous states have been hatched.

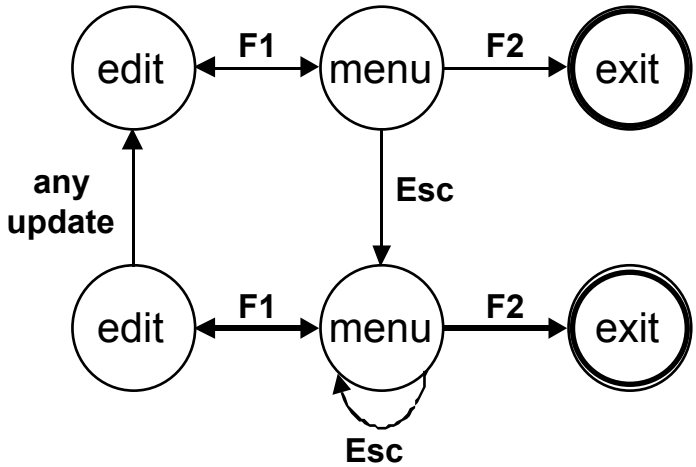

Figure 2.12 Revised STN with dangerous states

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 2.13 — disaster!

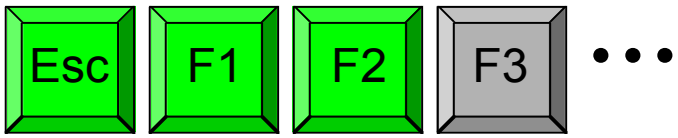

Figure 2.13 Dangerous function key layout

2.5 Summary

Because it forces you to be explicit using a formal specification forces the designer to clarify design issues. Details which might be missed or ignored are made explicit and salient. We saw that even when discussing a simple four function calculator and it will also be apparent in the next chapter. This does not mean that one is swamped in a sea of detail. The ability to work at different levels of abstraction means that you can choose which aspects require this precise treatment. However, having selected the appropriate abstraction, the formalism does not allow you to fudge issues!

Generic models are designed to allow particularly high level analysis. In our example, we saw how a formal analysis showed that certain types of undo command are impossible. It would be virtually impossible to come to this conclusion without a formal analysis. Even many failed attempts at designing and programming such an undo system would not convince one of its impossibility as the complexity of the task would suggest that there were still alternatives to try. However, note that the result of this analysis is an improved understanding of undo which can then be communicated without necessarily using full formalism.

We saw that formal modelling techniques, although powerful and useful, require a high level of formal expertise. The generation of informal understanding is one way that the benefits of formal work can distributed. However, in order to 'give away' the benefits of this work to the typical human-factors practitioner less maths' intensive forms of analysis are also 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).

Diagrammatic dialogue notations are not less formal because they are graphical. They simply are formal about different things. Even a drawing of a proposed screen design is formal in that one can check the eventual system against this 'specification': are the specified fields present, are they positioned as shown? Similarly, an hierarchical task analysis, as in Chapter 7, makes very precise and formal statements about the performance of a task using a system. Any form of specification involves some formal parts about which it makes precise statements, some informal parts perhaps in the form of textual comments, and some things about which it says nothing. The important question is not whether or not you should use formal methods – instead one needs to look at the methods being used and ask whether you know which aspects are formal, and if so whether they are being formal about the right things. And if not – well perhaps you should try some of the ideas in this chapter!

Further reading

General

Dix, A. J., Finlay, J. E., Abowd, G., and Beale, R (1993). *Human-Computer Interaction*. London: Prentice Hall.

The material in this paper draws extensively from Chapters 8 and 9 of this book, which also expands upon several of the areas.

Formal models of interaction and specification

Dix, A. J. (1991). *Formal methods for interactive systems.* London: Academic Press.

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

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

An edited collection covering a range of formal techniques.

Thimbleby, H. W. (1990). *User Interface Design.* New York: ACM Press, Addison-Wesley.

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

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

The original PIE paper.

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

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

Dix, A. J. (1992). Beyond the interface. In *Engineering for Human-Computer Interaction*, Larson, J. and Unger, C. (eds.), pp. 171–190. North-Holland.

Describes some aspects of status/event analysis relating status/event phenomena to the timescales over which they operate and the concept of *pace*. See also Chapter 9 of (Dix et al., 1993) for status–event timeline diagrams and Chapter 10 of (Dix, 1991) for its formal roots.

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 (Dix, 1991) and Chapter 12 of (Thimbleby, 1990).

Abowd, G. D. and Dix, A. J. (1992). Giving undo attention. *Interacting with Computers* 4(3), 317–342.

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

Archer, Jr., J., Conway, R. and Schneider, F.B. (1984). User recovery and reversal in interactive systems. *ACM Transactions on Programming Languages* 6(1), pp. 1–19.

A classic paper analysing different forms of undo.

Vitter, J. S. (1984). US&R: A new framework for redoing. *IEEE Software* 1(4), pp. 39–52.

Takes undo and redo to its extreme!

Yang, Y. (1988). Undo support models. *International Journal of Man-Machine Studies* 28(5), pp. 457–481.

Informal analysis and review.

Dialogue

As well as the following, see Chapter 8 of (Dix et al., 1993) which describes dialogue properties in more detail and any book on UIMS.

Alexander, H. (1987). Formally-based Tools and Techniques for Human-Computer Dialogues. London: Ellis Horwood.

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

Thimbleby, H.W. (1993). Combining systems and manuals. In *HCI '93: People and Computers VIII,* J.L. Alty, D. Diaper and S. Guest (Eds.) pp. 479-488. Cambridge: Cambridge University Press.

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

References

Abowd, G. D. (1990). Agents: communicating interactive processes. In *Human–Computer Interaction – INTERACT'90*, Cockton, G. and Shakel, B. (eds.), pp. 143–148. Elsevier Science Publishers.

Abowd, G. D. (1991). *Formal Aspects of Human–Computer Interaction*. Technical Monograph PRG-97, Oxford University, Programming Research Group, D.Phil. thesis.

Abowd, G. D. and Dix, A. J. (1992). Giving undo attention. *Interacting with Computers*s 4(3), 317–342.

Alexander, H. (1987). *Formally-based Tools and Techniques for Human-Computer Dialogues.* London: Ellis Horwood.

Archer, Jr., J., Conway, R. and Schneider, F.B. (1984). User recovery and reversal in interactive systems. *ACM Transactions on Programming Languages* 6(1), pp. 1–19.

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

Dix, A. J. (1991). *Formal methods for interactive systems.* London: Academic Press.

Dix, A. J. (1992). Beyond the interface. In *Engineering for Human-Computer Interaction*, Larson, J. and Unger, C. (eds.), pp. 171–190. North-Holland.

Dix, A. J., Finlay, J. E., Abowd, G. D., and Beale, R (1993). *Human-Computer Interaction*. London: Prentice Hall.

Dix, A. J. (1994). LADA a logic for the analysis of distributed action. In *Proceeding of the Eurographics Workshop on the Specification and Design of Interactive Systems*, Carrara, Italy.

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

Johnson, C. W. (1992). *A Principled Approach to the Integration of Human Factors and Systems Engineering for Interactive Control System Design*. YCST 92/05, University of York, Department of Computer Science, D.Phil. thesis.

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

Thimbleby, H. W. (1990). *User Interface Design.* New York: ACM Press, Addison-Wesley.

Thimbleby, H.W. (1993). Combining systems and manuals. In *HCI '93: People and Computers VIII,* J.L. Alty, D. Diaper and S. Guest (Eds.) pp. 479-488. Cambridge: Cambridge University Press.

Thimbleby, H. W. (1993). *Literate using for finite state machines.* University of Stirling.

Vitter, J. S. (1984). US&R: A new framework for redoing. *IEEE Software* 1(4), pp. 39–52.

Yang, Y. (1988). Undo support models. *International Journal of Man-Machine Studies* 28(5), pp. 457–481.