CHAPTER 1

Introduction

1.1 Formal methods and interactive systems

This books looks at issues in the interplay between two growth areas of computing research: formal methods and human–computer interaction. The practitioners and styles of the two camps are very different and it can be an uneasy path to tread.

1.1.1 Formal methods

As the memory of computers has increased, so also has the size and complexity of the software designed for them. Maintaining and understanding these systems has become a major task. Further, the range of tasks under direct control of computers has increased and the effects of failure, in say a space station or a nuclear power plant, have likewise increased. There may be little or no opportunity for direct control if the software malfunctions, for instance, in a fly-by-wire aircraft. Further, the costs of producing software have increased dramatically, and the possibility of maintaining code has diminished. This cumulation of factors has been termed the *software crisis* (Pressman 1982) and has led to a call for software design to become more of an engineering discipline. In particular, there is the desire for a more rigorous approach to software design, possibly including elements of mathematical formalism and having the possibility of being proved or at least partially checked.

Several varieties of formal methods have been developed for different purposes:

- *Graphical methods* – Such methods include Jackson Structured Programming (JSP) (Jackson 1983) for the design of data processing (DP) systems and various dataflow methods (Yourdon and Constantine 1978). Typically, only a portion of the required information is held in the graphs. The rest may be informally annotated or there may be additional non-graphical

notations, as is the case with JSP. Again the notation may be self standing, or be part of a larger methodology.

- *Program proof rules and semantics* – Another strand of formality concentrates on proving properties of programs or program fragments, (Hoare 1969, Dijkstra 1976) implicitly defining a meaning for the language. Others search for more explicit expression of program language semantics. (Stoy 1977)
- *Specification notations* – These are languages and notations specifically designed for the formal specification of software. Examples include Vienna Definition Method (VDM), (Jones 1980) Clear (Burstall and Goguen 1980) and Oxford's Z notation (Sufrin *et al*. 1985, Morgan 1985). Often these notations have explicit rules for the correct transformation of a specification towards an implementable form. (Jones 1980)
- *Methodologies* – Instead, or as well as defining precisely how a design is to be specified or proved, there are many methodologies aimed at defining what should happen in the design *process*. For example, JSP, mentioned above, is part of a complete design process. These formal methodologies may incorporate parts of the design process that are beyond what could be expected of an entirely rigorous approach. These approaches may involve graphical and textual notations and may be amenable to computer verification of consistency (Stephens and Whitehead 1986).

There is an underlying assumption to much of this book that software is being developed using some formal notation of the third category. Various sections of the book propose parts of a semi-formal design methodology.

1.1.2 Interactive systems

In the early days of computing the modes of interaction with the user were severely limited by the hardware available: initially cards and switches, later teletypes. The more limited the capabilities the greater the need for effective interface design, but early users were usually experts and there was little spare processing power for frills. Even as processing power increased and the interface hardware improved there was still a strong pull from the experts, who were the major users, for power and complexity. Two major influences have pushed the computer industry towards improved user interface design:

- *Technology push* – The realisation that there was the possibility of a computer society prompted research using state-of-the-art technology into futuristic scenarios. The Xerox work on Smalltalk (Goldberg 1984) and the Star interface (Smith *et al*. 1983) are examples of this.

- *Large user base* – The plummetting cost of hardware has led to a huge growth in computers in the hands of non-computer-professionals. The personal computer boom has taken the computer out of the hands of the DP department, and the new users are not prepared for inconsistent and obscure software.

The two strands are not independent. The Xerox Star has led to the very popular Macintosh, and the WIMP (windows, mice and pop-up menus) interface has become standard in the market-place. If one were to balance the two, it is perhaps the latter strand which is really of most significance in the current high prominence of issues of human–computer interaction (HCI).

The perceived importance of HCI is evidenced by the large number of conferences dedicated to it: the CHI conferences in USA, HCI in Britain, INTERACT in Europe, and HCI International. The "man–machine" interface was also a major strand of the Alvey initiative (Alvey 1984) and (under a different name) of its successor, the IED program. HCI also has a prominent role in the European Community's Esprit program.

1.1.3 The meeting

There is a certain amount of culture shock when first bringing together the concepts of formal methods and interactive systems design. The former are largely perceived as dry and uninspiring, in line with the popular image of mathematics. Interface design is, on the other hand, a more colourful and exciting affair. Smalltalk, for instance, is not so much a programming environment as a popular culture. Also it is hard to reconcile the multi-facetedness of the user with the rigours of formal notations. Some of these problems may be to do with misunderstandings about the nature of formalism (although even I, a mathematician, find a lot of computer science formalism very dry). However, this is not a problem just between formalisms and users: mathematical and formal reasoning typically is performed by people and does therefore have a more human side. The shock really occurred when living users met dry unemotional computers and must therefore be dealt with in any branch of HCI. However, the gut reaction still exists and is a reminder of the delicate balance between the two.

No matter how strong the reaction against it, there is clearly a necessity for a blending of formal specification and human factors of interactive systems. If systems are increasingly designed using formal methods, this will inevitably affect the human interface, and if the issue isn't addressed explicitly the methods used will not be to the advantage of the interface designer. If we look again at some of the reasons for needing formal methods, large critical systems where the crisis is most in evidence clearly need an effective interface to their complexity. The penalty for not including this interface in the formal standards will be more

accidents due to human error such as at Chernobyl and Three Mile Island, and the more powerful the magnifying effect of the control system the more damaging the possible effects.

The need for more formal design is seen also in more mundane software. Many of the problems in interactive systems are with awkward boundary cases and inconsistent behaviour. These are obvious targets for a formal approach.

1.1.4 Formal approaches in HCI

There are several approaches taken to the formal development of interactive systems:

- *Psychological and soft computer science notations* – These include the layered approach of Foley and van Dam (1982), or the more cognitive and goal-oriented methods such as TAGPayne 1984.lf 1718 - The uses of these vary, for instance improving design, predicting user response times and predicting user errors. They are not intended for combination with the formal notations of software engineering.
- *Specifying interactive systems in existing notations* – Several authors use notations intended for general software design to specify interactive systems. Examples of this include Sufrin's elegant specification of a text editor using Z, (Sufrin 1982) a similar one by Ehrig and Mahr (1985) in the ACT ONE language, and no less than four specifications in a paper by Chi (1985) in which he compares different formal notations for interface specification. Sometimes it is some component of the interface that is specified rather than an entire interactive system, as is the case with the Presenter, an autonomous display manager described by Took (1986a, 1986b, 1990). Pure functional languages have also been used to specify (and implement) interactive systems. Cook (1986) describes how generic interface components can be specified by using a pure functional language and Runciman (1989) has developed the PIE model, described later in this book, in a functional framework.
- *Notations for specification* – A general-purpose notation is not necessarily best suited to specifying the user interface, and various special purpose notations have been developed for interface, and especially dialogue, design. Hekmatpour and Ince, for instance, have a separate user interface design component in their specification language EPROL (Hekmatpour and Ince 1987). Marshall (1986) has merged a graphical interface specification technique with VDM in order to obtain the best of both worlds. Alexander (1987a, 1987b) has designed an executable specification/prototyping language around CSP and functional programming.

- *Modelling of users* – Another strand of work concerns the formalisation of the user. This may take the form of complex cognitive models using techniques of artificial intelligence, such as the expert system for interface design described by Wilson *et al*. (1986). Another proposal is *programmable user models,* an architecture for which programs can be writen that simulate the use of an interface. The approach is advocated by Young *et al*. (1989) with the intention of studying user cognitive processes. It has also been advocated by Runciman and Hammond (1986) and Kiss and Pinder (1986) with the aim of using the complexity of the user programs to assess the complexity of the interface.
- *Architectural models* – Any specification or piece of software has some architectural design, and specific user interface architectures have been designed with the aim of rationalising the construction of interactive systems and improving component reuse. These may be structuring techniques for existing languages such as PAC (Coutaz 1987) (Presentation–Abstraction–Control) an hierarchical agent-oriented description technique, or the MVC (ch) (Model–View–Control) paradigm used in many Smalltalk interfaces; or may be part of an overall system as is the case with UIMS (Pfaff 1985) (User Interface Management Systems). Architectural techniques are often combined with notations for dialogue design and (more rarely) interface semantics. Production rules, for example, are frequently used as the dialogue formalism in UIMS. On the other hand, interface design notations may implicitly or explicitly encourage particular architectural styles.

More extensive reviews of these different areas can be found in Alexander's thesis (Alexander 1987c) and in a report on formal interface notations and methods produced collaboratively between York and PRG Oxford. (Abowd *et al*. 1989) A recent collection of essays on the subject of formal methods in HCI edited by Harrison and Thimbleby (1989) contains papers in most of the above categories.

An additional category has become characteristic of the "York approach" to HCI, which is the main subject of this book:

- *Formal, abstract models of interaction* – These are formal descriptions of the *external* behaviour of systems. They are not models of specific systems, but each covers a *class* of interactive systems, enabling us to reason about and discuss interactive systems in the abstract. As well as a large body of work originating in York, (ck) the approach has been taken up by Anderson (1985, 1986), who uses a blend of formal language and denotational semantics to describe interactive systems, and by Sufrin and He (1989), who cast in Z, a model similar to the *PIE* model presented in Chapter 2.

We can lay out the formal approaches to interactive systems design in a matrix classified by concreteness and by generality (*fig*. 1.1). The concreteness axis distinguishes between the internal workings of the systems and the specification of their external behaviour. The former are more useful for producing systems, the latter for reasoning about them. The generality of a method may lie between those which can be realised only in the context of a specific system and those that have some existence over a class. Laid out like this, it is obvious that abstract models fill a crucial gap.

	generality	
concreteness	specific	generic
specification	notations for specification *task and goal descriptions*	abstract models
implementation	prototypes of the actual system *programmable user models*	architectural models *cognitive architectures*

figure 1.1 *formal methods matrix*

In drawing up the matrix (and making my point!) I have rather overplayed the gap filled by abstract models. Specifications of particular systems may be deliberately vague in places, and thus begin to encroach on the generality barrier. Similarly, architectural models, although aimed at implementation, may be given a suitable form for us to use for specification and reasoning, and hence begin to move up towards the domain of formal models. Cockton's work (cl) is a good example of this. He uses a description technique drawing on an analysis of UIMS. The notation is used to express properties of interface separability and comes close in spirit to the idea of an abstract interface model.

From the other side, the abstract models in this book are supplemented by examples of specifications of parts of actual systems, hence bridging the generality barrier from their side. Also, especially in Chapters 8 and 9, there is a movement towards more architectural descriptions, that is, a movement towards concreteness. Abowd (cm) has produced a notation which attempts to sit in this middle ground between the formal models of this book and architectural models. It is, of course, no good describing useful properties of systems in a highly abstract manner, if these cannot be related to more concrete and specific situations, and thus these areas where the various techniques overlap are most important.

The more psychologically based formalisms sit rather uneasily in the matrix, but I have included them as, to the extent that they do fit the classifications, a similar gap is seen on their side. Now the abstract models we will deal with are primarily descriptions of the system *from* the user's point of view. (But definitely *not* in the language a typical user would use!) They do have then an implicit abstract, generic model of the user, purely because of the perspective from which they are drawn. It is though a rather simple model, and a more explicit model might be useful. On the other hand, I find myself feeling rather uneasy about the idea of producing generic models of users: individuality is far too precious.

1.2 Abstract models

We have seen that abstract models fill a niche in the range of available HCI formalisms, but we also need to be sure that it is a gap worth filling. We shall take a quick look at why we need abstract models, and at the philosophy behind them.

1.2.1 Principled design

There are many principles for the design of interactive systems. Some are very specific (e.g. "error messages should be in red") and others cover more general properties (e.g. "what you see is what you get"). Hansen (1984) talks about using user engineering principles in the design of a syntax-directed editor Emily. Bornat and Thimbleby (1986) describe the use of principles in the design of the display editor ded.

Thimbleby (1984) introduced the concept of *generative user engineering principles* (GUEPS). These are principles having several properties:

- They apply to a large class of different systems: that is, they are *generic*.
- They can be given both an informal *colloquial* statement and also a *formal* statement.
- They can be used to constrain the design of a given system: that is, they *generate* the design.

The last requirement can be met at an informal level using the colloquial statement – as was the case with the development of ded. While not superceding this, it seems that at least some of the generative effect should be obtained using the formal statements. The authors cited above who have specified particular interactive systems have proved certain properties of their systems, by stating the properties they require in terms of the particular specification and then proving

these as theorems. The same approach could be taken for GUEPS; however, there are some problems.

- *Commitment* – The statement of the principles cannot be made until sufficient of the design has been completed to give an infrastructure over which they can be framed. This means that the principles cannot be used to drive this early part of the design process, which lays out the fundamental architecture, and hence may be crucial for the usability of the system.
- *Consistency* – Because the principles are framed in the context of a particular design, there is no guarantee that a given informal principle has been given equivalent formal statements in the different domains.
- *Conflict* – Expanding on the last point, there is a conflict between the desire for *generic* principles and the requirement for *formal generative* principles.

Formal abstract models can be used to resolve this conflict. The principles required can be given a formal statement using an abstract model. Because they do not represent a particular interactive system, but instead model a whole class of systems, we can reason about the effects and interactions of these principles as they apply to the whole class. Then, when a particular system is being designed, the abstract model can be mapped at a very early stage onto the design – or even refined into the first design. The principles defined over the abstract model can then be applied to the particular system.

We see that this tackles the problem of commitment because the principles are stated before the particular system is even conceived, and the problem of consistency because the principles are only stated once and then the same statement is applied to many systems.

1.2.2 Meta-models

Why is it that existing notations cannot be turned to this purpose? If we use a standard specification notation, the principles are stated within the context of a particular design – with the drawbacks described above. A principle stated using an abstract model is a requirement of what sorts of specifications are allowable within the notation. That is, it is really a statement about the *meta-model*.

Of course, any notation has, either explicitly or implicitly, a meta-model within which it works. What is wrong with these? Couldn't they be used instead of designing specific abstract models for the principles? The problem is that the notations are designed for the complete specification of the interface and therefore do not satisfy the principle of *structural correlation* necessary for effective requirements capture. This is not to say that an abstract model cannot be expressed within a particular notation. Many general-purpose notations are quite capable of expressing their own meta-model, and therefore an abstract model is no problem. An example is Sufrin's use of Z for models similar to

those within this book. However, when this is done the abstract model is stated in the notation: we are not using the meta-model of the notation itself. The abstract models of this book are stated within the notation of general mathematical set theory, but can be translated into whatever notation is being used for a particular design.

1.2.3 Abstract models and requirements capture

Designing a system involves a translation from someone's (the client's) informal requirements to an implemented system. Once we are within the formal domain we can *in principle* verify the correctness of the system. For example, the compiler can be proved a correct transformer of source code to object and we can prove the correctness of the program with respect to the specification. Of course, what cannot be proved correct is the relation between the informal requirements and the requirements as captured in the specification. This gulf between the informal requirements and their first formal statement is the *formality gap* (*fig*. 1.2).

For a DP application like a payroll, this may not be too much of a problem. The requirements are already semi-formal (e.g. pay scales, tax laws) and they are inherently formalisable. The capture of HCI requirements is far more complex. Not only are they less formally understood to start with, but it is likely that they are fundamentally unformalisable. We are thus aiming to formalise only some aspect of a particular requirement and it may be difficult to know if we have what we really want.

If the abstract model is designed well for a particular class of principles, it can form a bridge between the informal requirements and the formal specification. To achieve this, there must be a close correspondence of structure between the abstract model and the informal concepts. This principle of *structural correlation* is an important theme.

1.2.4 Surface philosophy

The abstract models must reflect the structures of the requirements in order to bridge the formality gap. What does this mean for interactive systems and user-centred design? For particular classes of properties there will be particular structures of importance that must be reflected in the abstract model. For example, when considering windowed systems in Chapter 4, two informal characteristics are focused on, *contents* and *identity*. These are incorporated into the formal model. I will later argue that the identification of such informal characteristics is not only a precursor to formalism, but is part of the formal method itself.

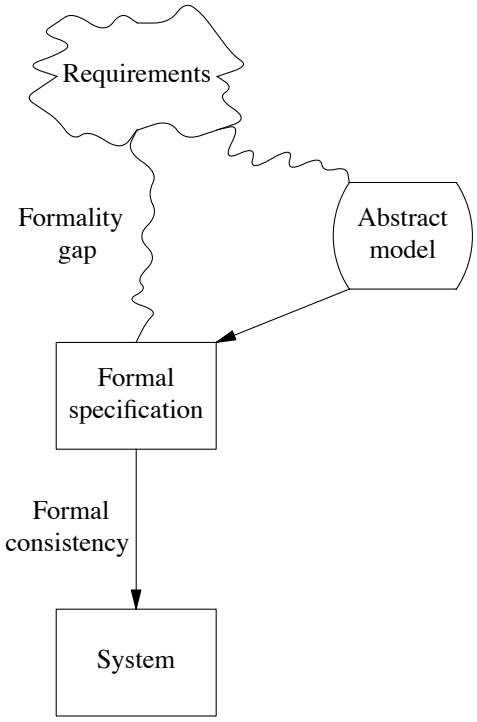


figure 1.2 *formality gap*

Not all the features of the abstract model are dependent on the particular principles required. If we are searching for a user-centred design, then we should not be concerned with parts of the system that are not apparent to the user. We should adopt a *surface philosophy* when designing abstract models. We are not interested in the internal details of systems, such as hardware characteristics, languages used, or even specification notations! The models will be, as far as possible, *black-box models* concerned only with the user's input, the system's output, and the relation between them defined as abstractly as possible.

This approach, prompted by our interest in the human-computer interface, corresponds exactly to the desire in formal specification not to overspecify or, as it is often put, to specify "what", not "how". Using a formal specification notation does not prevent overspecification. Jones (1980) warns that a specification may contain *implementation bias,* that is, it may accidentally specify some aspects of a system that should be left undetermined. He gives formal checks for this. Even if a specification satisfies these conditions, it may still have aspects which although not constraining the design totally, do tend to

push it in fixed directions. This is another example of *structural correlation*: the structure of the abstract model/specification strongly affects the eventual system.

Sometimes it may be necessary to break the black box slightly. Often the user may perceive some of the internal structure Where this is so, the model should reflect it. This may arise because the system is badly designed and you can "see the bits between the pixels". For instance, the user may be aware of implementation details, such as recirculating buffers or event polling, by the way the system behaves, or even worse be presented with error messages such as "`stack overflow`"! It may also occur in a more acceptable fashion. For instance, it may not be unreasonable for the user to be aware of the filing system as a separate layer of abstraction within an operating system.

Opening up the box does have dangers, however. Users' models may differ, and in particular they will differ from that of the designer. For example, the distinction between the operating system and a programming environment may not be evident to the novice. Even with the example cited above, of the filing system, there may be confusion when using an editor, between the editor's internal buffer and the filing system state. This may lead to errors such as removing a disk without writing the file. Because of these dangers, we will try to retain the black box view as far as possible, and only break it where it is absolutely necessary and when the user model is very clear.

This argument also leads us away from methods based around a user's goals, or cognitive models of the user. This is not because they are not useful. Applied as an evaluation or prediction tool on existing systems, they too complement the approach taken here. Just as the desire for a user-centred approach leads to a surface philosophy with respect to the computer, the need for a *robust* analysis leads to a similar view of the user.

When stating principles over the abstract models, care is again taken to ensure there is no overcommitment to a particular implementation technique or view of the system. So, for example, when we consider sharing properties of windows we reject *explicit* definitions of sharing using the data base or file system, and instead look for an *implicit* definition based on the input/output behaviour. Similarly, in Chapter 5, we define the buffering of user events not in terms of implied internal structures, such as blocks of memory, but in *behavioural* terms at the interface.

1.3 Formalities

So we are going to use formal methods to help us design interfaces. What do we mean then by formal methods, and what sort of results are we likely to get?

1.3.1 What are formal methods?

First and foremost, formalism is not about a particular notation. That much is obvious. Further, it is not the use of *a* notation or even general mathematical notation. When I think formally, εs, δs and Σs rarely cross my mind. These may be used for the communication or recording of the thought, but they do not constitute the formalism in themselves. Not only is this important in that we shouldn't demand such notations before we call something formal, but also just because we see mathematical notation, it is no guarantee of the soundness of the arguments. Even mathematically logical arguments can be spurious when related to the real-world entities they describe.

It is particularly common in computing circles to associate formal methods with a very strict and rigorous form of mathematical logic. The majority of mathematics operates at nothing like this level, always being prepared to leave gaps where the correctness is obvious. In a similar vein, Millo *et al.* (1979) argue that proof is essentially about raising confidence. This is in line with the argument presented earlier to justify abstract models as a bridge between requirements and formal specification. Also, I will later argue that structural correlation between abstract model and specification and between specification and implementation is crucial to raise confidence in the correctness of the final system – whether the relation is strictly proved or not.

So if proof is just a way of raising confidence, what of the *process* of formalisation? In essence, to formalise is to *abstract,* to examine features or parts of a whole and hence understand the whole better. This means that when in Chapter 4 I say that the critical aspects of a windowed system are content and identity, this is not an informal statement about to be formalised, but a formal statement about to be captured in precise notation.

Some of this abstraction can be captured in ordinary English, and indeed the language is rich in terms to describe abstractions of objects. So, for instance, I am a man, a human, a mammal, an animal and a living thing. I am also a biped and a mathematician. Common English is less good for discussing relations between objects in the abstract, and other higher-order concepts. In particular, it is very difficult to describe some of the real-time phenomena in Chapter 5.

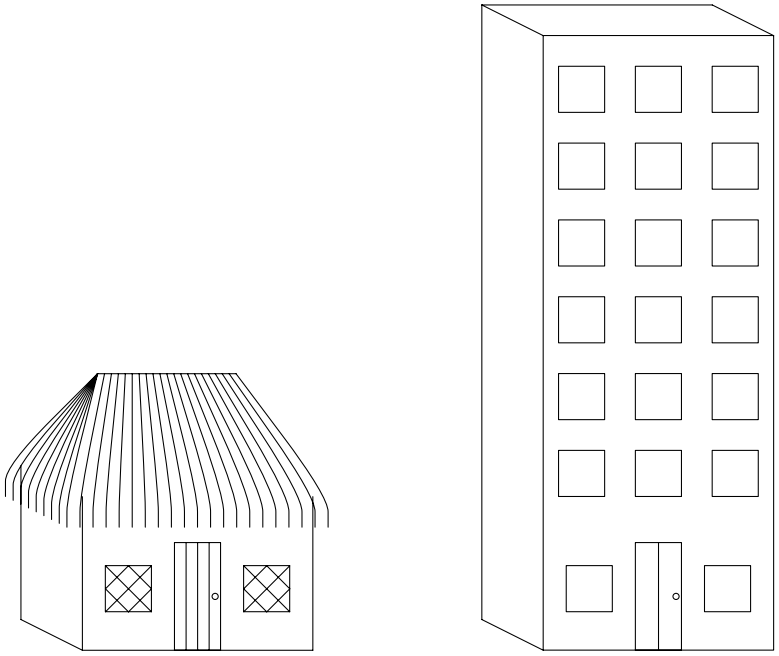
In short, I use mathematical notation because it is useful, but the formalism is not restricted to the mathematics and the mathematics does not guarantee formalism. Despite all this, and for want of a better term, I will usually talk of the precise mathematics as being formal, and common English statements as

being informal, although the boundary may not be clear.

1.3.2 Applicability of formal methods to design

Thinking of formalism in terms of precisely written rules and formulae, what is its range of applicability for interface design? If we asked instead more definitely – "Will we be able to generate a precise definition of usability?" – we could answer "NO" with little fear of contradiction. However, just where formal methods lie, between being useless and essential, is not obvious.

The use that is most obvious and most "formal" for the abstract models and principles defined in this book is as a *safety net*. We can see this if we look at the role of formalism in the design of a building. We have two putative designs, one a thatched cottage, and one a concrete block of flats.



Formal properties of building materials can tell us things like the thermal conductivity of thatch, whether a concrete lintel can bear the required weight, whether the structures are water-tight. These may be codified into formal requirements, either explicit like building regulations on insulation and drainage, or implicit – it mustn't fall down! The formal analysis does *not* tell us which is the best design. Even if we formalise more of the requirements for a particular context – How many people will live there? Are they scared of heights? – we

still have an incomplete picture. A formal method cannot tell us that uPVC windows on the cottage would be barbaric, or that leaded panes on the flats would be plain silly. So the formalism of building design can tell us whether the building will fall down or leak, but not whether it will be beautiful or pleasant to live in.

Going back to interface design, by defining suitable principles we will be able to stop interfaces being fundamentally unusable, but not ensure that they *are* usable. So we can't make a bad interface good, but we can stop it being abysmal. The challenge, of course, is to do this and still allow the good interface designer to be creative. That goes beyond this book, but what I have found is that the uses of abstract models for principled design go way beyond being just a safety net. However, even in that capacity only, the experience of other disciplines shows that they would still be worthwhile.

1.4 Editors

1.4.1 Universality of editing

Many of the motivating examples used in this book will be about text editing. Does this limit the applicability of the results?

I am not alone in this emphasis. Moran in a keynote address at Interact'84 referred to text editors as the "white rat" of interface design. Rasmussen at Interact'87 (Rasmussen 1987) similarly termed the word processor the "Skinner box", although he thought that HCI ought to move on from there. Despite this latter view, there are strong arguments for testing a new technique on an old field: if the technique proves useful despite much existing analysis, then this speaks highly in its favour.

I should like to extend that somewhat by saying that editing is universal. Almost all applications can be viewed as editing. For instance, an operating system can be seen as a file-system editor – compilers and other tools are just sophisticated commands. The drive towards direct manipulation emphasises this point still further. The exceptions to this view are non-closed domains like mail systems, but even they have a very large editing and browsing component.

Although editors provide motivating examples, the interaction models are intended to be quite general. This assertion is validated by the way the windowing model of Chapter 4 and the temporal model of Chapter 5 relate easily back to the general PIE model which was designed very much with editors in mind. Some of the facets will require a more general non-deterministic version of the PIE model, which would be suitable for a mail interface.

1.4.2 Some example editors

Occasionally I shall use some actual editor as an example. I describe briefly the nature of those used.

Ed (Thompson and Ritchie 1978) is the standard Unix editor. It is a line editor, like all line editors just more so!

Vi (Joy 1980) has become a standard display editor on Unix systems. It does not demand special keys of the terminal, but uses the normal typing keys in two main modes. Normally they mean some editing command, so for instance "k" means move up one line. If you ask to insert in a line, or add to the end of a line then there is a mode change and the keys generate characters, so "k" would insert as a character in the text. Even experts forget which mode they are in, and it is not uncommon for users to type editing commands into their text, or (more disastrously) type text which is treated as commands. This latter at best generates beeps and errors, and at worst destroys your text – hence the pun in Chapter 6.

Wordstar (Wordstar 1981) is the *de facto* standard word processor for personal computers. It reserves ordinary keys for their plain text meaning, and uses control key combinations for commands such as cursor movement. It has many inspired features, but seems to be looked on as rather a *bête noire* by the human interface community.

Ded (Bornat and Thimbleby 1986) is a display editor working under Unix. It inspired Sufrin's specification. Bornat and Thimbleby concentrated on the principles that underly its design, and on not simply adding features. It has a characteristic command line at the bottom of the screen, which is used for editing commands that cannot be mapped to single keystrokes.

Spy (Collis *et al*. 1984) is a multi-file editor for bit-mapped workstations. Its commands are all operated by menus or mouse clicks. It is based on the paradigm of a selected region of text, like most such editors.

1.5 A taste of abstract modelling

To give a flavour of what is to come we will take a quick look at a simple abstract model, designed to express the property of "what you see is what you get". This model, slightly reformulated, was my first use of an abstract model and it is offered as a first taste to the reader also. Although it is very simple it exposes a serious and pervasive problem, *aliasing*, and thus demonstrates the power of the technique.

1.5.1 A simple abstract model to express WYSIWYG

"What you see is what you get" (WYSIWYG) is one of the most well known principles of interface design. A large number of word processors, CAD systems, etc. claim to be WYSIWYG. Thimbleby (1983) points out that this is usually referring to superficial aspects such as having the correct fonts, whereas there is a much deeper principle expressing the way the system can be manipulated and viewed via its display. This he terms "what you see is what you *have* got". The principles I will try to formalise are based upon this notion rather than the presentation one.

An early suggestion to give his principle a precise formal statement in the context of simple text editors was to say an editor is "WYSIWYG" if the text it represents can be recreated as the concatenation of the various displays generated. This was only intended as a starter to more general definitions, being rather domain specific and restricting the possible displays to be simple windowing. Another problem is that it relates display to internal state, rather than describing the interface properties purely in terms of the external interface. This last point is easy to correct: the displays should (in the case of a text editor) be related to the final printed form. Depending on whether there is any formatting of output the map from internal state to printed form may be trivial or complex, so it is important that we retain the external view as much as possible. In fact, we could go further and express the relation between different facets of the interface in terms of the user's perception of them and the resulting internal models to which they give rise. From this we get a simple picture of the interface (*fig.* 1.3).

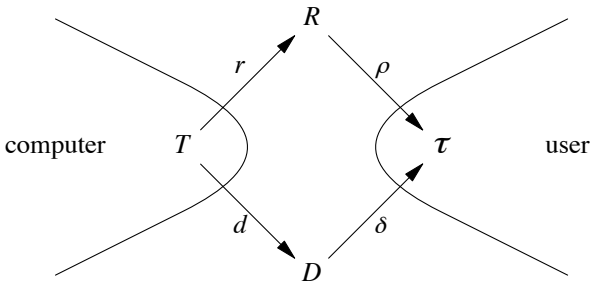


figure 1.3 *WYSIWYG model*

The internal representation of the object being edited (from T) can be printed (using the map r) to yield the result (from R) which is then viewed by the user (with a possibly noisy perception function ρ) to yield an internal idea of the object (from τ). Similarly, the object can be displayed (using d to generate a

display in D) and is also viewed (using δ). The viewing of the result is potential (you don't print the entire text after each command!), whereas the viewing of the display is continual. Further, although there is only one result for a particular object, there are many possible displays, so, strictly, the display should be parametrised over some set Λ, giving the set of displays $\{d_\lambda\}$.

This cannot be read as a simple commuting diagram, as a single display will not in general yield all the information about the object, just part. The map δ is not therefore a simple function but instead provides *partial information*. This partial information could be represented by the set of objects in τ consistent with a particular display, or more generally by a function to a lattice based on τ. We can then give simple definitions of WYSIWYHG in terms of the information ordering $\leq$:

consistency:
$$\forall\, \lambda \in \Lambda \;:\; \forall\, t \in T :\quad \delta \circ d_\lambda(t) \quad \leq \quad \rho \circ r(t)$$

That is, the information provided about the object from the display is consistent with the text that would be obtained by printing.

It is no good the display being consistent if it doesn't show you the bits you want to see (a blank screen is consistent with anything). We really want the stronger principle "what you can see is *all* you have got": that is, there should be nothing in the result that is not viewable via the display:

sufficiency:
$$\forall\, t \in T :\quad \bigvee_{\lambda \in \Lambda} \delta \circ d_\lambda(t) \quad = \quad \rho \circ r(t)$$

Clearly, since the individual $\delta \circ d_\lambda(t)$ are less defined than $\rho \circ r(t)$, then the least upper bound is also less well defined. Unfortunately, in even apparently innocuous cases equality is not obtained and unexpectedly stringent requirements are made on the system if sufficiency is required to hold.

1.5.2 Aliasing

Consider the following example. Let T be sequences of binary digits and the display (D) is either a pair of binary digits or a "before the start" symbol («) or an "after the end" symbol (»). The individual displays are obtained as adjacent pairs of digits from the object and all such displays are possible. The result map and perception functions are all identity. Thus t = "01" has the following possible displays:

"«0", "01", "1»"

In this case there is only one possible consistent object, that is, t itself. However, if t = "0100" then the possible displays are:

"«0", "01", "10", "00", "0»"

In this case not only is τ = "0100" possible, but so is τ = "0010" (and this is assuming the length is known!). This is the problem of *aliasing* and it will arise repeatedly in different guises: essentially, the *content* of a display does not unambiguously define the *context*. We can, of course, overcome this by explicitly coding the context in the display. For instance, in the previous example, we could augment the display by adding a position so that the displays would be:

"[0]«0", "[1]01", "[2]10", "[3]00", "[4]0»"

This time there is only one candidate, τ = "0100", as required.

This is not an artificial problem. For instance, large files of data gathered for statistical analysis may contain tracts of identical (or very similar) information. Again when editing programs, several functions may have very similar form and the user may mistake which function is being edited. This second example is more problematical, as in the former case the user is likely to be aware of the possibility of confusion, whereas in the latter, guards will be down.

Some editors do in fact display the line number (or byte number!) of the current cursor position, or alternatively a scroll bar with the current display highlighted presents the same information. However, it is not intuitively reasonable to make all systems display such contextual information, and later definitions of WYSIWYG properties will be given that are more liberal. Because of aliasing these definitions will be considerably more complex and lack the conceptual clarity of a purely information theoretic approach.

1.5.3 Implications for future models

The class of problems that considerations of WYSIWYG lead to I will call *observability* and will be a major interest when considering various models.

Another interesting point that is exemplified in this simple context is the conflict between pedagogic statements and more complex problems of perception. Consider the scroll bar. We might argue that the maximum document size that avoids aliasing is when each pixel of the scroll bar corresponds to a screen of text. More circumspectly, we realise that the user's perception (represented by δ) is unlikely to be perfect, and although it may be reasonable to regard it as identity when reading text, it is unlikely to be up to resolving differences of one pixel between successive displays. After further reflection, one realises that it is likely that users may ignore the positional information entirely unless they are consciously aware that aliasing may arise. So, it may well be that an editor with explicit contextual information loses this in

perception, whereas another editor with no explicit information, but which helps the user develop an internal sense of context, perhaps by allowing only small moves and emphasising direction by smooth scrolling, may be better.

If we continue to include the user's perception in our models, then our formal statements can include such considerations: however, this is unlikely to be totally correct even when considering a single user and certainly not for an entire user community. For this reason most formal statements will be made at the interface level (e.g. about d and r here), effectively assuming users are perfect perceivers and reasoners. This is in line with the "surface philosophy" applied to users and the limitations of formal methods described above. At various stages the system should be checked against more informal statements of principles (and/or formal models of perception). The two processes of formal and informal reasoning complement one another and yield a more robust total method.

1.6 About this book

The flow of chapters in this book reflects a partly historical, partly logical progress of different models and methods. The material ranges from highly formal to anecdotal, and may move rapidly between the two. For those without a rock-hard constitution a simple cover-to-cover reading is unlikely to be successful. I will give a brief overview and suggest ways of reading.

1.6.1 The landscape

The book has two main phases of model building. The first five chapters (2–6) deal with some fully abstract black-box style models; the primary purpose is to express usability principles for different contexts. Chapters 7–9 "open up" the black box slightly, reflecting the way users know more about a system than its physical surface. The last three chapters round off the book in various respects.

Principles – abstract models

We begin with what is in some ways the simplest and most general model, the PIE model. It is simple in that its mathematical formulation is sparse, and in that it assumes very little in the way of detail about the systems it describes. However, the analysis in Chapter 2 is probably the most mathematical and detailed in the entire book. By being very rigorous on the simplest model, we are freed to take a slightly looser approach in other areas knowing that we may perform similar activities if it were required.

A good example of this is the use of state-based models. The PIE model takes a very external view of the system. Not only does it not describe the state of the system, it shies away from even mentioning the state! One important result that

comes out from the analysis is that it is always possible to give a state-based description of a system that is no less abstract than such a behavioural one. Further, for any view of the system, we can obtain an effective state (called the *monotone closure*) which captures all that's needed to predict the future behaviour of that view. In later chapters, we use state-based formulations where appropriate in the knowledge that we can assume this state to be the minimal "behavioural" one.

The next chapter is about red-PIEs. These augment the very sparse PIE model by distinguishing the display (what you see) from the result (what you will get). The properties of interest are primarily issues of observability: what can we tell about the result from the display. The key phrase is the "gone away for a cup of tea" problem. I use my computer, come back, and wonder where I've got to. The chapter investigates what I can infer about the system upon my return.

Multiple windowed systems are increasingly common. Chapter 4 suggests a model that is capable of describing the special behaviour of such systems. It is particularly concerned to address issues of interference between windows. This is important because of the possibility that different windows are concerned with different tasks and therefore interference between windows means interference between tasks.

In common with many interface descriptions, the basic red-PIE model describes the interleaving of input and output, but not detailed temporal behaviour. The model defined in Chapter 5 can be used to describe behaviour such as *buffering* and *intermittent* and *partial update*. In order to counter some of the problems raised by real-time behaviour it proposes that certain information is available to tell the user when the system is in *steady state* and when commands will be ignored. The model is so constructed that steady-state behaviour can be specified independently of the real-time behaviour.

Chapter 6 tries to bring some of the models together, but finds that a *non-deterministic* model of interaction is required. This leads on to a discussion of the meaning and role of non-determinism in the interface. It is found to be a useful descriptive technique and unifying paradigm. An example of its constructive use is the proposal of a new efficient display update strategy.

Where appropriate these chapters discuss the implications of the principles discussed. For instance, Chapter 5 discusses the various forms that the information on real-time behaviour might take depending on where the user's focus of attention is. It also discusses the requirements that its proposals put on support software such as operating systems and window managers.

Design – opening up the box

Chapter 7 begins this phase by focusing on the design of layered systems: where the inner layers represent the functional core or task activities of the system, and the outer layers the physical level such as keyboards and screens. It investigates various constructive ways of building up the entire system given the inner functional core. Two major conclusions arise. The first is that the relationship between the functional core and the entire system is one of abstraction rather than of being a component. The second is that any understanding of manipulative systems must include a very close connection between the interpretation of users' actions and the presentation of the system's responses. The user's input is mediated by the display context. As a sort of last fling at a constructive model Chapter 7 looks at the use of *oracles* in designing and prototyping systems.

Chapters 8 and 9 take two possible routes to display-mediated interaction. Chapter 8 looks at *dynamic pointers*. Position is regarded as the crucial feature in the interpretation of user commands in a display-oriented system. This leads on to a detailed investigation of the properties of positional information in interactive systems. Dynamic pointers are thus aimed at translating the user's display-level commands into operations on the underlying objects. Chapter 9 takes a complementary approach. It is assumed that users' primary perception is manipulating the view they have of the system. The important issue then becomes how to change the underlying state to keep it consistent with the user's view. The rather odd conclusion of this chapter is that when updating using views, it is what you do *not* see which is of central importance.

Rounding off

Chapter 10 focuses on events (things that happen) and status (things that are). This is partly as an admission of how poorly the majority of the models presented here deal with this distinction, and partly as an attempt to rectify the situation. It also forms a link to the study of multi-person interaction, in contrast to the one-user-one machine assumption of most of the book.

The models and methods in this book aren't much good if they cannot be applied to real situations. The most important application is the way that such models open up new ways to think about interactive systems. I hope that readers will find this for themselves as they move through the book. Chapter 11 gives more explicit examples of how the models can be applied. These range from "informal" applications of the formal models in a real situation, to the way they integrate into a formal development paradigm of software engineering.

Finally, Chapter 12 reflects back on the models presented and some of the threads that emerge using abstraction as a unifying theme.

Appendices – notation

Appendix I describes the notation used in this book and also how to read function diagrams such as the one we have already seen in §1.5 and found in various places, especially in Chapter 3. Appendix II contains a specification of a simple text editor which is analysed in Chapter 11.

A little history

The PIE model is the starting point both historically and logically of many of the other models in this book. It has also become the key term associated by many with the whole spectrum of models and techniques generated at York. I considered writing the relevant chapter for this book using a slightly different formulation, but decided that any gain in comprehensibility of such a new perspective would be outweighed by the confusion for those familiar with the original formulation. In addition, I would have been in grievous danger of losing the acronym entirely!

1.6.2 The route

As I have said, it is probably a bad idea to start at the beginning and work through. The chapter on PIEs is the logical starting point but is one of the most complex mathematically. In order to ease this somewhat I have tried to summarise the primary points raised by the chapter in its introduction. This can be read in order to give the reader enough background to move on to other chapters. In general, I would encourage the reader to skip parts which they find too formal (but **not** *vice versa*). Dependencies are typically loose enough that later parts need only a general understanding of previous material, remembering always that it is the concepts that matter, not the particular formulations.

Quite a lot of the chapters can be read on their own. In particular, Chapter 5 on temporal behaviour has appeared on its own elsewhere (cs) and is quite readable on its own. Also, a slightly extended version of Chapter 6 appeared in (Harrison and Thimbleby 1989) and, if you are prepared to believe that non-deterministic models occur naturally when considering formal models, the latter part of it can also be read on its own. The second phase of the book, "opening up the box", requires familiarity only with the PIE model (and red-PIE), and Chapters 8 and 9, although naturally leading on from this, are each self-contained. Chapters 10 and 11 both refer back to previous chapters, but both are largely readable with perhaps a passing knowledge of earlier material.

For the reader who doesn't want to get bogged down in heavy formalisms too early I would suggest starting with Chapter 5, and (with the caveat above) Chapter 6, most of Chapter 10, and the first two sections of Chapter 11. For the more confident reader ... well I'll let you decide.