CHAPTER 6

Non-determinism as a paradigm for understanding the user interface

6.1 Introduction

Although this chapter primarily presents conceptual and pragmatic ideas about non-determinism in user interfaces there is another, higher-level, more important point to it. This chapter provides an example of the subtle interweaving of formal and informal reasoning: the way formal analysis can lead to new insights which influence the informal.

We start with an analysis of problems arising from several formal models. In addition to the PIE model we started with, we have now defined two further models, one for windowed systems and one for temporal systems. These models have obvious similarities, and it would be useful to relate them to one another. The first section of this chapter attempts to do so, and quickly we discover that in order to do this we require non-deterministic models to express properties of interest. These models are non-deterministic in a purely formal sense; the rest of the chapter considers the repercussions non-determinism has on our understanding of the user interface.

Non-deterministic models contradict the general feeling that computer systems are deterministic, following a fixed sequence of instructions, so we might wonder whether there is any meaning to this use of non-determinism, or whether it is mererly a useful, but essentially meaningless, formal trick. In the rest of this chapter, we will demonstrate four things about non-determinism in user interfaces:

- It does exist.

- Users deal with it.
- We can help them do this.
- We can use it.

Sections §6.3 and §6.4 deal with the first of these points. The first of these shows that non-determinism is in fact a common experience of users, and we focus on the notion of *behavioural* non-determinism in order to square this with the normal deterministic model of computation. The second section then catalogues various sources of non-determinism in the user interface.

After this we consider how users, abetted by the designers of systems, can deal with this non-determinism. The final point of §6.5 reminds us how non-determinism can be useful in the specification of systems; however, this non-determinism is not intended to be a facet of the actual system, only a tool for development. Section §6.6 goes beyond this, giving an example of how non-determinism might be *deliberately introduced* into the user interface to improve performance. Without being prepared by noting how supposedly deterministic systems behave as if they were non-deterministic, the deliberate introduction of non-determinism would seem preposterous. Instead we are able to assess it impartially and see how it may actually *reduce* apparent non-determinism.

6.2 Unifying formal models using non-determinism

So far in the book, we have dealt with three major interaction models:

- *PIEs* (including *red-PIEs*) – a very general model intended to be applicable to almost all systems.
- *Handle spaces* – for modelling windowed systems.
- *A temporal model* – for considering real-time properties of interactive systems.

In any particular system, we may want to apply properties expressed over several of these models and we could simply map them all onto the system independently; however, it is clearly sensible to consider how they interrelate at the abstract level, and if possible relate them all back to PIEs, which are intended as the general model.

For temporal models we have already seen an example of this, where we abstracted the steady-state functionality and described how it fits into a general design method. However, even there we will find other abstractions we would like to consider.

In the following section we will consider such unification of these models and how it leads us to consider non-determinism. The first two subsections look at temporal models and handle spaces respectively, and consider how these relate to PIEs. In each case we will find that the PIE is insufficient and that a non-deterministic model is necessary. We will have to generalise the PIE model, giving a non-deterministic PIE (ND-PIE). This will come in versions with and without languages, and, in contrast to the deterministic PIE, the version with the language (non-deterministic language!) is the more natural and requires extra limitations to remove the language. We then return to the temporal and windowed models and relate them to the ND-PIE. We find that several properties can be expressed as determinacy requirements on certain derived ND-PIEs. Having considered these two models which started the investigation we look again at the deterministic PIE and see whether any abstractions of this can usefully be described using non-determinism. Again we find that predictability can be seen as a determinacy requirement.

6.2.1 Problem for temporal systems

In the previous chapter we saw how a PIE representing the steady-state functionality of a system could be embedded into a fully temporal description of that system. As we noted, a real user of the system would encounter a more complex functionality depending on the exact timing of input. However, the user will be unaware of such detailed timings (with a clock operating at MHz!), and it is only some abstraction of this temporal behaviour which is apparent. Clearly, from any sequence from C_τ we can abstract a sequence from C by simply ignoring all the τs. This is the user's view of the input (i.e. what was entered but not exactly when). It is not clear what effect to associate with a particular input sequence but for the moment let's use the steady-state effect.

Having removed timing from the temporal system the result is likely to be non-deterministic. For instance, take the bufferless typewriter which requires a single tick after each character in order to reset itself and ignores any characters being entered too fast:

$$\begin{array}{lcl} C & = & \textit{characters} \\ E & = & \textit{sequences of characters} \end{array}$$

$$\begin{array}{lcl} I_\tau(\ \textit{null}\ ) & = & \textit{null} \\ I_\tau(\ c\ ) & = & c \\ I_\tau(\ p\tau\ ) & = & I_\tau(\ p\ ) \\ I_\tau(\ p\tau c\ ) & = & I_\tau(\ p\ )\ c \\ I_\tau(\ pc_1c_2\ ) & = & I_\tau(\ pc_1\ ) \end{array}$$

Thus the input sequence "abc" with different timings may give rise to different effects:

$$I_\tau(\ a\tau b\tau c\ ) \quad = \quad abc$$
$$I_\tau(\ ab\tau c\ ) \quad = \quad ac$$

Clearly, we need a non-deterministic version of the PIE model to model such behaviour.

6.2.2 Problem for windowed systems

A similar problem arises when considering the handle space representation of windows from Chapter 4. Clearly, we can regard the whole system as a PIE with a command set of $C \times \Lambda$ and with a language defined by the valid handles map. More often we would be interested in regarding each window as an interactive system in its own right.

If we "freeze" all other windows we can give the functionality of a window λ in the state e as the result and display interpretations I_r and I_d:

$$I_r \quad = \quad result \circ I_\lambda$$
$$I_d(\ p\ ) \quad = \quad display(\ I_\lambda(\ p\ ), \lambda\ )$$

where I_λ is the iterate of *doit* relative to λ:

$$I_\lambda(\ null\ ) \quad = \quad e$$
$$I_\lambda(\ pc\ ) \quad = \quad doit(I_\lambda(\ p\ ), c, \lambda\ )$$

This definition is not very useful: it's not very interesting knowing the functionality of a window system when you only use one window! What we really want to know is the functionality of each window when other windows are being used also. To do this we would consider the possible effect of commands p to a window amidst all possible interleavings from other windows. If all commands are result and display independent in all contexts, then this would yield the same interpretation functions as above. In the general case, however, we would again obtain non-deterministic functionality.

6.2.3 Non-deterministic PIEs

We will now consider non-deterministic generalisations of the PIE model. There are many ways of modelling non-determinism, but one of the simplest is to substitute for some value a set of possible values. However, we have to be careful to choose the right representation of our model, and the right values to substitute. The simplest option is to assign to each input a set of possible effects.

That is, we modify the signature of the interpretation to $I_{ND}: P \rightarrow \mathbb{P}E$, where $\mathbb{P}E$ is the collection of sets of effects. Unfortunately, this does not distinguish some systems that are clearly distinct. Consider I_{ND} defined as follows:

$$\forall p \in P \quad I_{ND}(\ p\ ) = \{\ 0, 1\ \}$$

Does this describe a system that starts off with a value of 0 or 1 and retains this value no matter what the user enters? Does it represent a system that makes an independent choice after each command? Is it something in between? On the basis of the information given, we cannot decide. We therefore need to consider the *trace* of all effects generated by a command.

In Chapter 2, we saw that for any deterministic PIE we can always define a new interpretation $I^*: P \rightarrow E^*$ by:

$$\begin{array}{lcl} I^*(\ null\ ) & = & [\ null\ ] \\ I^*(\ pc\ ) & = & I^*(\ p\ ) :: [\ I(\ c\ )\ ] \end{array}$$

where "::" is sequence concatenation. That is, we define an interpretation giving the entire history of effects for each command history. This is the appropriate function to generalise for non-determinism yielding an interpretation $I^*_{ND}: P \rightarrow \mathbb{P}E^*$. We could then distinguish the single random constant system with interpretation:

$$\forall p \in P \quad I^*_{ND}(\ p\ ) = \{\ zeroes, ones\ \}$$

where *zeroes* is a sequence of zeroes of length $length(\ p\ ) + 1$ and *ones* is a sequence of ones of the same length. From the multiple independent-choice system:

$$\forall p \in P \quad I^*_{ND}(\ p\ ) = \{\ 0, 1\ \}^{n+1}$$

where $n = length(\ p\ )$.

We can see that necessary conditions for a valid interpretation I^*_{ND} are:

effect history is right length for number of inputs:

$$\forall e^* \in I^*_{ND}(\ p\ ) \quad length(\ e^*\ ) = length(\ p\ ) + 1$$

history cannot change:

$$\forall q \leq p \in P, e^*_p \in I^*_{ND}(\ p\ ) \quad \exists e^*_q \in I^*_{ND}(\ q\ ) \quad \textbf{st} \quad e^*_q \leq e^*_p$$

where $\leq$ is the initial subsequence relation. The first condition says that the effect trace must contain one member for each input command plus an initial effect, the second that if some sequence of effects has been the result of a sequence of commands then its first $m + 1$ effects must be a possible result of the

initial m commands.

From now on we will call a triple $< P, I^*_{ND}, E >$ satisfying these conditions a non-deterministic PIE, abbreviated ND-PIE.

We can say that a ND-PIE is deterministic if for all p there is at most one effect given by $I^*_{ND}(p)$. That is:

$$\forall p \in P \quad \| I^*_{ND}(p) \| \leq 1$$

If any of the $I^*_{ND}(p)$ are empty then the resulting PIE has a language.

If we don't want our ND-PIEs to have input languages, then we have to put more restrictions on I^*_{ND}. It is insufficient in the general case simply to ask for I^*_{ND} always to be non-empty. Consider I^*_{ND} where:

$$\begin{array}{lcl} I^*_{ND}(\text{"}ab\text{"}) & = & \{ \text{"000", "111"} \} \\ I^*_{ND}(\text{"}abc\text{"}) & = & \{ \text{"0000"} \} \end{array}$$

If we had typed "ab" and got the series of responses "111", then there is no valid response if we typed a further "c". That is, the ND-PIE accepts a non-deterministic input language. The proper additional rule to prevent this is to ensure that for any input p there are possible extensions to this, no matter what additional input we type:

$$\forall p \leq q \in P, e^*_p \in I^*_{ND}(p) \quad \exists e^*_q \in I^*_{ND}(q) \quad \textbf{st} \quad e^*_p \leq e^*_q$$

where $\leq$ is again the initial subsequence relation.

Note the way this is a dual to the "history cannot change" condition. Whether we want such a condition is debatable: it depends on the level of system description we are using and on whether there are any fundamental constraints to user input (like typing at non-existent windows, or trying to use a bank-teller when its cover is down). In fact, each of the ND-PIEs we are going to derive will satisfy this property.

6.2.4 Use for temporal systems

We can now use the ND-PIE to represent the non-deterministic functionality we required in §6.2.1. That is:

$$I^*_{ND}(p) \quad = \quad \{ I_\tau{}^*(h) \quad | \quad \xi(h) \; = \; p \}$$

where ξ is the function extracting the user commands from a sequence containing ticks:

$$\begin{array}{lll} \xi(\ null\ ) & = & null \\ \xi(\ h\tau\ ) & = & \xi(\ h\ ) \\ \xi(\ hc\ ) & = & \xi(\ h\ )\ c \qquad c \neq \tau \end{array}$$

With this definition the example of the typewriter which misses characters typed too quickly gives us:

$$\begin{array}{lll} I^*_{ND}(\ abc\ ) & = & \{\ [\ null, a, a, a\ ],\ [\ null, a, a, ac\ ], \\ & & \quad [\ null, a, ab, ab\ ],\ [\ null, a, ab, abc\ ]\ \} \end{array}$$

Not only can we now define this functionality, but it gives us a new way to look at the system. In Chapter 5, perfect buffering was defined. Looking at the definition of I^*_{ND} we see that perfect buffering is precisely the requirement that I^*_{ND} is deterministic.

6.2.5 Use for windowed systems

We consider using a window (with handle λ) whilst ignoring possible interleaved commands to a second window (λ'). This yields a projection from the handle space onto an ND-PIE:

$$\begin{array}{lll} I^*_{ND}(null) & - & \bigcup_{q \in P} \{\ [doit^*(e, q, \lambda')]\ \} \\ I^*_{ND}(pc) & - & \bigcup_{e^* \in I^*_{ND}(p),\, q \in P} \{e^* :: [doit^*(doit(e^*, c, \lambda), q, \lambda')]\ \} \end{array}$$

where $doit^*$ is the natural extension of $doit$ to all members of P. Again we can use this to give an alternative definition for a user interface property. Result independence between λ and λ' is precisely the condition that $result \circ I^*_{ND}$ is deterministic.

6.2.6 Non-deterministic properties of PIEs

We have seen that properties over temporal models and handle spaces can be given statements as determinacy properties of ND-PIEs. Are there any interesting ND-PIEs that can be abstracted from simple PIEs?

In Chapter 2 we considered the simple predictability property that a PIE is monotone if:

$$I_p(\ null\ ) = I_q(\ null\ ) \quad \Rightarrow \quad \forall s \quad I_p(\ s\ ) = I_q(\ s\ )$$

where I_p was the interpretation function "starting" with a command history p. We examined this in the context of the "gone away for a cup of tea" problem, where one has forgotten exactly what command sequence had been entered before the cup of tea. This suggests non-determinism about the value of p, and we can consider the ND-PIE generated by this:

$$I^*_{ND}(\ s\ ) \quad \equiv \quad \{\ {I^*}_p(\ s\ )\ \}_{p \in P}$$

If this ND-PIE is deterministic then the PIE is rather uninteresting, as its functionality is independent of the commands entered. It is a deaf system! The predictability condition can be stated using this ND-PIE as:

$$\begin{array}{l} \forall s \in P,\ e^*_1,\ e^*_2 \in I^*_{ND}(\ s\ ) \\ \quad \mathit{first}(\ e^*_1\ ) \ = \ \mathit{first}(\ e^*_2\ ) \ \Rightarrow \ e^*_1 \ = \ e^*_2 \end{array}$$

This is a measure we could apply to any ND-PIE whether or not it is derived in this way. It is quite a strong requirement, saying that although the system is non-deterministic, one glance at the first effect resolves all future doubt. We could, of course, go on and give non-deterministic equivalents to the more refined concepts of predictability, for instance defining strategies over ND-PIEs.

One other ND-PIE suggested by the definition of I^*_{ND} above is if we substituted arbitrary PIEs for the collection I_p. For example, given two interpretations I and I' over the same domains P and E, we could define the non-deterministic interpretation:

$$I^*_{ND}(\ p\ ) \quad \equiv \quad \{\ I(\ p\ ),\ I'(\ p\ )\ \}$$

If this ND-PIE is deterministic, then I and I' are identical. Later, in §6.4.4, we will see a real situation that could be described using this.

A specific case is where the interpretations are PIEs representing the "same" system with different start data. If we consider the red-PIE, we will often have the case where there is a "bundle" (tray!) of PIEs, each indexed by an element of the result, $\{\ I_r\ \}_{r \in R}$. Each PIE starts with the appropriate result, and is related to the others by:

$$\begin{array}{l} \mathit{result}(\ I_r(\ \mathit{null}\ )\ ) \quad = \quad r \\ \forall r,\ r' \in R \quad \exists p^{r'}_r \in P \\ \quad \textbf{st} \quad \forall p \in P \quad I_{r'}(\ p\ ) \ = \ I_r(\ p^{r'}_r \ p\ ) \end{array}$$

The ND-PIE generated from these interpretations:

$$I^*_{ND}(\ s\ ) \quad \equiv \quad \{\ {I_r}^*(\ s\ )\ \}_{r \in R}$$

represents non-determinism about the starting value of the system. Again we will see later how this arises informally.

6.2.7 Summary – formal models and non-determinism

We have seen how a non-deterministic model has been useful in unifying the description of various properties in diverse models. The various examples share one common feature: in each case, the deterministic model is viewed via an abstraction which corresponds to losing some part of the available information. This leads to non-deterministic behaviour. In each case the non-deterministic model used to capture this behaviour was the ND-PIE; however, this is largely because the various models were derivatives and extensions of the basic PIE model. Different flavours of model could be dealt with similarly using different non-deterministic models and perhaps using different methods of expressing the non-determinism.

When viewed in the light of the previous discussion, properties such as predictability and non-interference of windows become efforts to control non-determinism. Either they assert that in certain circumstances the effect is deterministic, or give procedures to resolve it.

6.3 Non-deterministic computer systems?

In the last section, we found that formal models of non-determinism are useful for describing certain abstract properties of interactive systems; however, we were left wondering whether this formal construction held any meaning for the user. We accept that the internal workings of some systems will be non-deterministic, especially where concurrent processes are used, and even that some specialist applications like simulations and certain numerical methods will involve random number generation, but surely most real systems have a deterministic external interface?

6.3.1 The tension for the user

Do users perceive computers as deterministic? In fact, the opposite is the case: most users expect a degree of randomness from the systems they use. Time and again they will shrug their shoulders in bewilderment, "Oh well, it didn't work this time, I'll try the same thing again, it may work now." The apparently random behaviour of such systems conflicts with the alternative model of the computer as a deterministic machine relentlessly pursuing its logical course. Some users are able to cope with this. Expert users may treat it as a challenge, puzzling over the behaviour and experimenting until a logical reason is found. Pragmatists will accept the occasional strangeness and circumvent it, while the awestruck user will regard it as part of the magic and mystery of modern technology. Others may have a more negative reaction. The self-confident may react "this is silly" and lose all confidence in the computer. The self-depreciating

may respond "I'm silly", attributing their problems to their own lack of understanding, and possibly retiring from further use. Phrased in these graphic terms the problem seems extreme and demands investigation, but how can it be that thoroughly deterministic programs give rise to this apparent randomness?

6.3.2 Levels of non-determinism

Very few systems are really random: even random number generators are usually based on deterministic algorithms, ERNIE (a random number generator based on quantum effects used in a national lottery in the UK) being a possible exception. Programs that rely on external events could be classified as non-deterministic, for instance the time when a printer signals that it has emptied its buffer, but even then the printer itself will probably behave deterministically. In fact, what is usually termed non-determinism reflects the things that the programmer either doesn't know or doesn't want to know. In other words, it is programmer centred.

We can classify non-determinism according to the level at which it appears:

- *Mechanistic world* – The atomic events measured by ERNIE and the actions of people could certainly be regarded as non-deterministic, and are incapable of prediction even when considering the whole of the computer system. Even here, whether these events are *really* non-deterministic can be argued; however, we are getting into the realms of metaphysics.
- *Computer* – Other events, like printer signals, are deterministic when we take the entire mechanism of computer and printer together, but are not so from the point of view of the computer alone – unless it has a very sophisticated model of the printer, in which case it needn't bother with handshaking at all!
- *Programmer* – The scheduling of programs within a multi-programming system is deterministic from the computer's point of view since it is applying some scheduling algorithm (round-robin, priority-based, etc.). From the programmer's point of view, however, this is non-deterministic and real time may jump suddenly between adjacent program steps. Similarly, file systems may change apparently non-deterministically as other programs operate.
- *User* – Even totally deterministic calculations such as "is $579217^{11} - 1$ prime?" are non-deterministic from the user's point of view, and are effectively the same as if the computer had tossed a coin to find the answer. Many systems do not use such obviously obscure formulae but manage to produce interfaces that are equally bizarre.

The theme that comes out of the above is that non-determinism is relative: relative both to knowledge and to reasoning abilities.

6.3.3 Behavioural non-determinism

What should a user-centred view of non-determinism be? Imagine two systems which each have two possible prompts, and each day they choose a different prompt for the day. One system bases its choice on whether the number of days since 1900 is prime or not, the other on the decay of a slightly radioactive substance. From the programmer's point of view (and the computer's), we would say that the former is deterministic and the latter not. From the user's point of view the two systems display equally non-deterministic behaviour.

The user sits at the bottom of the hierarchy of knowledge: all the forms of non-determinism are equally random, and should be treated equivalently. That is, we are going to take a *behavioural* view of non-determinism. This recognises that some systems may be "really" deterministic and others may be "really" non-deterministic according to some definition, but if they appear to behave the same, we will regard them as equally non-deterministic. Not only does this mean that we regard some "really" deterministic systems as non-deterministic, but also *vice versa*. For instance, if we had a music system with some random "noise" that led to errors of 1 part per million in the frequency of notes, we would regard the system as being deterministic, as the difference in behaviour would be undetectable.

We could demand tighter views of non-determinism, but for the purposes of this chapter we will adopt the behavioural one. Again, one could use a weaker word to represent behavioural non-determinism; however, the use of such a charged word concentrates the mind wonderfully.

6.4 Sources of non-determinism

In the last section we decided that, taking a behavioural view of non-determinism, it did make sense to describe interface behaviour in these terms. In doing so, we introduced some examples to argue the point. In this section we will catalogue informally some of the sources of non-determinism in the user interface, giving more examples on the way. We will study these sources of non-determinism under six headings:

- Timing
- Sharing
- Data uncertainty
- Procedural uncertainty
- Memory limitations

- Conceptual capture

The first two of these cover the informal equivalents of the two formal problems that started this chapter. The second two consider problems that appear even in the steady-state functionality of single windowed systems, and cover lack of knowledge about *what* you are dealing with and *how* you should do it. The last two are to do with the more complex ways that human limitations can give rise to apparent non-determinism. These two could be thought of as "the user's fault", but the system designer cannot wriggle out of it so easily; good system design should take into account the limitations of its users.

I am sure this list is not complete; however, it gives a broad spectrum of different types of non-determinism to which the reader can add.

6.4.1 Timing

Several of the problems noted in Chapter 5 can be viewed as manifestations of non-determinism. When users type quickly, intermittent and partial update strategies produce different outputs than if they had typed more slowly. If the user is unaware of this exact timing, then the system's behaviour is apparently non-deterministic. This non-determinism is usually deemed acceptable since the final display does not depend on the exact timing, and, further, spells of fast typing may be regarded as single actions anyway. Intermittent update could be said to be less non-deterministic than partial update, since at least all its intermediate displays would have arisen with slow typing. On the other hand, a portion of the screen in partial update is always exactly as it would be if the machine were "infinitely fast", and this portion is therefore totally deterministic.

Similarly, the problems associated with slow machines and buffering can be thought of in terms of non-determinism. A machine that doesn't buffer and loses characters typed too quickly could be regarded as non-deterministic on this score. This is exactly the non-determinism captured by the formal model at the beginning of this chapter. However, a system with buffering can lead to a non-deterministic feedback loop between user and computer, leading for instance to cursor tracking.

Scheduling of multiple processes leads to non-determinism. If for some reason a system makes explicit or implicit use of real time, then, when running on a time-share computer, the run times of its components, and hence possibly its functionality, will be non-deterministic. More often than not this dependency on real time will be in the assumptions made about the relative speeds of certain components, or in the assumption that two statements following one another will be executed immediately, one after the other. The most innocuous case of this is when several concurrent processes print messages to the terminal in random order.

6.4.2 Sharing

Again, the problem of sharing can be regarded as one of non-determinism, reflecting exactly the formal treatment. For instance, as I transact with one process which is my focus of interest, other processes may print error messages on the screen. Because I am preoccupied with the focal process, I may have forgotten that the others were running and thus be temporarily confused. In this case I would not remain confused for long, as I would remember what other processes were running (or ask the system), and infer their behaviour. Thus the non-determinism could be resolved, but not soon enough to stop me acting (potentially disastrously) on a changing system.

This situation is in the middle of the three levels of sharing discussed in Chapter 4. Each of these types of sharing can lead to non-determinism:

- *Single actor – multiple persona* – The user is simultaneously involved (perhaps via windows) with several dialogues, which may share data. When switching from one dialogue to another, he may forget that actions in one dialogue may have repercussions in the other, thus causing apparent randomness. Literally the right hand may not know what the left is doing.
- *Single controller – several actors* – The user sets off several concurrent processes that affect data common to each other and to the user's current view. This is the original case given above, and differs from the single-actor case in that changes may actually occur as the user is actively involved in a dialogue, rather than during interruptions to the dialogue. Because of this it is apparently more non-deterministic, since the system is changing as the user tries to manipulate it.
- *Several independent actors* – This is the case of the multi-user system where not only are several things happening at once, but they are not under a single user's control. This is the most random of all, since the machine processes are at least in principle predictable, but from each user's point of view the other users are fundamentally non-deterministic processes. On the other hand, this is the case where working sets are least likely to overlap, and where protection mechanisms are most likely to exist.

6.4.3 Data uncertainty

A user has a program on a system which was written a long time ago, and the exact contents of it are now forgotten. He wants to make changes to this program and invokes his favourite editor by entering the command "edit prog". This is an example of *data uncertainty*; the user does not know the exact value of the data on which he is going to operate, and a major goal of the interactive session is to reduce the uncertainty, perhaps by scrolling through the file to examine it. This corresponds to the ND-PIE generated from the bundle of PIEs

in §6.2.6.

Data uncertainty is, of course, common. The example given is typical of uses over many different types of task. We would not usually class it as a problem: we do not expect to know all the data in a computer system by heart. The importance lies in the user's ability to discover the information: that is, in the *resolution* of the non-determinism, a point to which we shall return in the next section.

6.4.4 Procedural uncertainty

Consider the following two situations:

- A user is within a mail utility and has just read an item of mail which is only 10 lines long and is still completely visible on the screen. She wants her reply to include a few selected lines from the message and does this by using the mail's "e" command in order to edit the text of the message. There are several editors on the system. "I wonder which it will use", she thinks.
- The user who is editing the program (from the example in the previous subsection) decides that the variable names "a,b,c" are not very evocative of their meaning and decides to change them to day-total, week-total and year-total, respectively. He then realises he has forgotten the method of achieving a global search/replace.

In the first situation, the exact nature of the data is known and it is *procedural uncertainty* from which the user suffers. She will look for clues using the knowledge she has of editors on the system. The editor fills the entire screen so it can't be a line editor like "ed", she surmises. Neither can it be mouse-based, like "spy", for it doesn't have menus all over the place. It could be either of the screen editors "ded" or "vi". Tentatively she types in a few characters to see if they're inserted in the text. The editor beeps at her a few times, then deletes half the text... now she knows it's definitely "vi". This form of procedural uncertainty is captured by the ND-PIE which chooses between a set of interpretation functions. It is interesting that the two situations which appear quite different informally, have such a similar informal definition.

The second case is another example of procedural uncertainty of a less extreme and more common form!

6.4.5 Memory limitations

As stated previously, one of the causes of non-determinism is lack of knowledge. This is exacerbated by the fact that people forget. Thus as the user attempts to amass information in order to resolve the non-determinism, her efforts are hampered by her limited memory. A designer must consciously

produce features which take account of this. These could include general features, such as an online memo-pad and diary, or more system-specific ones. Further, the user is likely to interpolate the gaps and be unaware of which information is known and which is inferred. This process is distinct from forgetting; we can think of it as *degradation of information*. The designer may need be aware of where such degradation is likely and actually force this information to the user's attention.

6.4.6 Conceptual capture

We are all familiar with the idea of capture, where for example one walks home without noticing when one intended to go to the railway station in the opposite direction. This occurs in computer systems where commonly used sequences of keys can take over when one intends to use another similar sequence. A similar process can also occur at the conceptual level. For example, in a display editor where long lines are displayed over several screen lines, the CURSOR UP key might mean move up one screen line or one logical line. As most logical lines will fit on one screen line the difference may not be noticed. Later when a long line is encountered the action of the system may appear random to a user who has inferred the wrong principle.

6.4.7 Discussion

Of the six headings which we've considered, the first four can be be given formal expression, to some degree or other, using the models developed in this and previous chapters. The last two would require a more sophisticated model of human cognition, with its attendant problems of robustness. There is an obvious parallel between data uncertainty and degradation of information, and between procedural uncertainty and conceptual capture. However, the second of each pair is far more dangerous. The major problem with both these situations when compared to procedural and data uncertainty is that the user may not be aware of the gaps in her knowledge. The situation where a user doesn't know something and *knows* she doesn't know it, is still non-deterministic but the situation where the user thinks she knows what the system is going to do and then it does something completely different, is downright random. We could say that failure in knowledge is far less critical than failure in meta-knowledge.

6.5 Dealing with non-determinism

We have seen that non-determinism is a real problem in the user interface, and that it has many causes. How can we deal with the problems of non-determinism? We will consider four options in this chapter; we can:

- Avoid it.
- Resolve it.
- Control it.
- Use it.

We will consider each of these options in turn.

6.5.1 Avoid it

The most obvious way of dealing with non-determinism is to make sure it never arises. This can be done by designing a system with this in mind, or by adding functionality afterwards. We have already seen examples of this:

- *Timing* – We have said that timing leads to non-determinism. However, the information suggested in Chapter 5 to tell the user when the system is in steady-state and when commands will be ignored, will reduce or remove this non-determinism.
- *Sharing* – Using the definitions of independence we could demand that systems have sharing properties that avoid non-determinism. Alternatively, we could use one of the responses suggested in Chapter 4 to make the sharing apparent, and hence reduce the non-determinism when interference occurs.

In both these solutions, we add information to the interface in order to avoid non-determinism. This is, of course, a general technique restricted only by the display capacity. For instance, if procedural uncertainty is the fault of hidden modes, then we can make the modes visible via a status line.

We can attempt to avoid conceptual capture either by ensuring the models we use are exact matches of the user's model or by making discrepancies very clear. This is, of course, very difficult advice to follow as it is difficult to predict what model the user will infer for the system. However, systems that propose a model (such as the desktop) but fail if the user follows it too far are obvious candidates for improvement. Another situation to be avoided is where two sub-subsystems have apparently similar semantic behaviour, but later diverge. If several possible models have similar behaviour during normal activities we should consider adding features to distinguish the particular model used.

In most systems of any complexity it would be unreasonable to expect complete removal of non-determinism; however, these techniques can reduce the non-determinism or remove some aspect of it.

6.5.2 Resolve it

Assuming the user is in a situation of non-determinism he can try to resolve that non-determinism, attempting by observation or experiment to reach a deterministic situation. Data uncertainty is an obvious candidate for this, and the concept of *strategies* introduced in Chapter 2 can be thought of as an attempt to resolve the non-determinism. When applied to *result predictability* it is precisely the resolution of data uncertainty. The use of the strategy for full predictability can be seen as the resolution of data uncertainty about both the result and internals such as cut/paste buffers. In addition it resolves issues such as mode ambiguity, which is a form of procedural uncertainty.

The designer can help the user in the resolution of data ambiguity by reducing the conceptual and memory costs of strategies. This is aided by the fact that the user will only want to discover some part of the information. Typical techniques include:

- *Improved navigation aids* – By easing the location of information the designer reduces the effort required of the user and makes it more likely that she will be able to remember sufficient information for the task at hand. Further, the job of refreshing memory can be significantly reduced. Mechanisms for this include, depending on the application, search commands in text editors, dependency trees in programming environments, cross-referencing and indices.
- *Place holders* – Because of degradation the user will need continual refreshing of information. Uncommitted navigation aids can be augmented by the ability to lay marks at important positions to refer back to, comparable to bookmarks.
- *Multiple windows* – In a similar vein, a system may allow several simultaneous views, thus making dispersed information simultaneously available. Owing to the limited size of displays all the views will not be present at one moment, and thus we may regard windows as a form of sophisticated place holder.
- *Folding mechanisms* – If the data are structured in a relevant manner, folding mechanisms can significantly aid navigation. Further, if the user has sufficient control over what is and is not folded then unwanted information can be folded away and only the relevant information made visible. Effective folding can thus satisfy some of the requirements for the place holders. It should be noted, however, that in order to achieve these aims, either the user will require control of the structure of the folding, or the fixed

structure must be very well chosen (arguably no fixed scheme would satisfy all needs).

Procedural uncertainty is more difficult to resolve. Help systems can be thought of as a way of resolving procedural uncertainty; however, they do of course have to make major assumptions about how much the user knows already. The one sort of procedural uncertainty that the help system definitely cannot resolve, is the method for invoking help. This underlines the need to use consistent and obvious rules for this (permanent icon, dedicated labelled key or the use of "h" or "?").

6.5.3 Control it

Rather than try to remove non-determinism completely, we can try to control it so that the non-determinism experienced is acceptable in some way. This can obtain at the local or the global level. That is, we can ask for complete determinism over a part of the system, or merely that some rules always apply for the system as a whole. We consider the local level first.

It is said that you ought to be able to use a system with your eyes shut. Extending this analogy a bit, we could observe that blind people are able to navigate a familiar room quickly and confidently, whereas they would use a totally different strategy for navigating a busy street. Similarly, when using a computer conferencing facility one might be able to touch-type because the layout of the keys is fixed, and attention is fixed on the screen where unexpected changes will occur. Thus in computer systems, as in real life, we need a solid base of determinacy in order to be able to concentrate on those areas where non-determinacy will occur. We call this requirement *deterministic ground*. Record- and file-locking facilities are an example of how we might for a period enforce determinacy on a shared domain. Similarly, protection mechanisms offer a more permanent way of ensuring some level of determinacy. However, if we look at the three levels of sharing, we see that only the multi-actor case is helped by having private files. So if I have several windows, or have some background jobs running, they are all assigned to the same user, and hence the file system protection would fail to protect me from myself. As an example of this breakdown of security, consider the semantics of the line printer spooler. In some systems, the print command does not make a copy of the file to be printed in a system spool area, but merely makes a note of the request and the file to be printed. The user, however, after issuing the command may reach closure on that operation and then go on to modify or even delete the file to be printed. Perhaps protection mechanisms could be extended to include files private to tasks and windows?

At the global level the effects of sharing are controlled by the accepted procedures that are used for updating them. Some of these are enshrined in the software, and some in organisational and social conventions. For instance, today my bank's teller machine might read £300; however, tomorrow it may read only £20. If I hadn't kept a close tally on my spending, I might not have expected the change and I would regard it as essentially non-deterministic; however, I would have enough faith in the banking system to believe the change to be due to some of my cheques clearing. If this belief were not widely held the non-determinacy of bank balances would become unacceptable and the banking system would collapse. Similarly, early in a day a travel agent may notice 5 seats free on a particular flight, but later in the day, on trying to book one for a customer, the request might be refused. The conclusion drawn is that someone else has booked the seats in the meantime. Thus we rely on the semantics of others' transactions to make the apparent randomness of our view of data acceptable. If the changes we observe in the data are not consistent with our understood semantics we will lose confidence in the data.

I would argue therefore that in order to understand the problems of sharing from a user-oriented view, we should concentrate on defining the semantically acceptable non-determinism of a system. That is, we are prepared to accept *limited non-determinism.* We can apply this to other fields, for instance, if we look at buffering strategies. A perfectly buffered system may well be non-deterministic because the screen does not reflect the current state of affairs; however, we might regard this as acceptable. In contrast, a word processor which ignored all typeahead would be regarded as exhibiting unacceptable non-determinism. The predicate describing perfect buffering is a limit to the non-determinism sufficient to make it acceptable. Similarly, we may not be too worried about the exact strategy a text editor uses when rearranging the display when the cursor hits the screen boundaries. It would be unacceptable if the cursor movement caused part of the document to be altered (I have used a text editor which did exactly this!). Again, the limitation that the cursor movement does not alter the document is sufficient (with others in this case) to make the non-determinism acceptable.

6.5.4 Use it

We have just argued that to a large extent, controlled non-determinism can be acceptable non-determinism. Every user manual uses this fact, as it defines only the external behaviour of the system. So, for instance, different versions of a word processor could use different algorithms, be written in different programming languages and even run on different hardware (in the same box!). Similarly, formal specifications define only the interface behaviour, leaving the internals undetermined. Thus all specification is a use of non-determinism. It is usually assumed that the systems described by formal specifications will be

deterministic; it is just which particular consistent system is chosen that is non-deterministic. One could certainly have non-deterministic systems which satisfy a given loosely defined specification, but this is not usually done. There is a paradox here: in order to make accurate statements about a system (and hence reduce non-determinism about its behaviour), specifications are used which are themselves non-deterministic.

There is also a strange duality of non-determinism within the interface. The computer system must, if it is to be useful, be non-deterministic. A completely deterministic system would never tell the user anything that the user didn't know already. Not very useful! On the other hand, from the computer's point of view, the user is non-deterministic, but if the user were not so, the system would always produce the same result. To some extent there is conflict between the two partners' search for determinism, and the programmer usually has the upper hand, forcing the user to answer in specified ways and in specified orders. From the programmer's point of view this could be thought of as producing a more deterministic response from the user. From the user's point of view this is at best restrictive, and possibly may seem non-deterministic because the programmer's arbitrary decisions may have little relevance at the interface. Thimbleby (1980) calls this excessive control by the programmer *over-determination*, and elsewhere I have proposed a technique for returning control of the dialogue sequence back to the user. (cd) Not surprisingly, this leads to programs with more non-deterministic semantics and which regard the user as a non-deterministic entity.

We can distinguish two aspects of interface non-determinism based on the previous discussion. First are those that are part of the application, and are what makes the application useful. Second are those in the interface, resulting from arbitrary decisions and complexity, which are not wanted. On the other hand, we could use this difference to make the distinction between application and interface, as desired for instance by Cockton (1986), by saying that the application is what we want to be non-deterministic and the interface is what we want to be deterministic.

6.5.5 Summary – informal analysis

We have seen how the user helped by the designer can avoid, resolve and control the non-determinism of interactive systems. We have also seen that non-determinism can be used in formal specification. Earlier we asked whether non-determinism existed at the user interface or whether it was just a formal trick, and decided that it is a real, meaningful phenomenon. We could parallel that now and ask: is the use of non-determinism just a formal trick for interface specification or can it really be used to improve user interfaces? The next section will seek to answer that question.

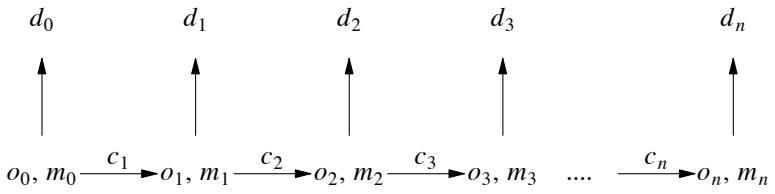
6.6 Deliberate non-determinism

So far we have dealt with cases where non-determinism has unintentionally arisen in the interface, and we have been interested mainly in removing it and its problems. In the last section, we saw that when developing systems, non-determinism can actually be used to good effect. In this section, we will consider a display update strategy that is deliberately non-deterministic. We will call this strategy *non-deterministic intermittent update*. It has considerable advantages over deterministic strategies from the point of view of efficiency; however, can such a deliberate policy of non-determinism ever be acceptable?

We will begin by describing a basic semiformal framework in which we can consider update strategies. Next we will consider the expression in this framework of total and intermittent update strategies (as described in Chapter 5). We then extrapolate these strategies and define non-deterministic intermittent update. Finally, we consider whether or not it is an acceptable strategy from the user's point of view, and conclude that by deliberately adding non-determinism to the interface we may actually *reduce* the perceived non-determinism.

6.6.1 Static and dynamic consistency

Many interactive systems can be considered in the following manner: there is an object (*obj*) which is being manipulated, and an associated screen display (*disp*). The way such a system is implemented is often as a state $< obj, map >$ consisting of the object and the additional information (*map*) required to calculate the current display. For example, if the object were a text then *map* may be the offset of the display frame in the text. As the user issues a sequence of commands (which may be keypresses, menu selections or whole line commands, depending on the system), the state changes in a well-defined and deterministic manner. Thus as a sequence of commands is issued we get a sequence of objects O_i mapping states m_i and displays d_i:



The earlier discussion on specification would lead us to question: what are the *requirements* for this update sequence? Typically they fall into two categories:

- *Static consistency* – At any stage we expect there to be a well-defined relation between the object and the display. For instance, in the case of text with a cursor position and a simple character map display with its cursor, we would expect the characters to match if we overlaid the two aligning the cursors.
- *Dynamic consistency* – Properties should hold between successive displays. Typical of this is the principle of display inertia, which says that successive displays should differ as little as possible.

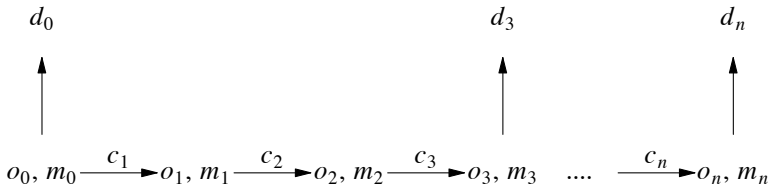
The designer will (more or less) have an idea of which static and dynamic requirements are important for a particular system; however, these will rarely specify the map completely and thus additional *ad hoc* requirements will be added to define it uniquely. (This is idealised, as often in practice these fundamental design decisions are made as the system is coded, with little reference to global impact.) The additional requirements may themselves be either static or dynamic.

Bernard Sufrin's specification of a display editor (Sufrin 1982) deliberately leaves the specific update strategy only partly defined; instead, he supplies a static consistency requirement and a dynamic "inertia" requirement that if the new cursor fits on the old display frame then the frame doesn't change. Many editors follow this rule of thumb and add additional rules to specify fully the case when the cursor does not fit on the old screen, such as "scroll just enough to fit in the cursor", "scroll a third of a screen in the relevant direction" or "centre the cursor in the new display". All these rules have additional special cases at the top or bottom of the text, and in the case of the first two, when the movement of the cursor is gross.

6.6.2 Intermittent update

As already noted, the time taken to compute the new map and update the display may lead to apparent non-determinism for the user, such as cursor tracking. Clearly we want to avoid this non-determinism.

If, after all other optimisations have been tried, the response is still unacceptable, a possible course is to use intermittent update as described in the last chapter. This corresponds to surpressing some of the displays in the sequence:

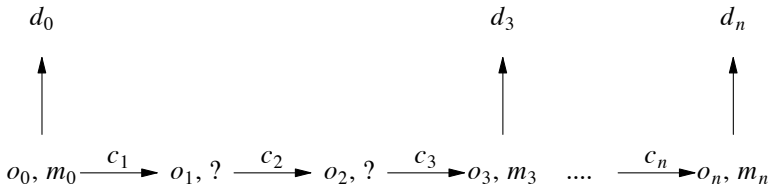


This is mildly non-deterministic, in that the sequence of displays is different depending on exactly how fast the user types. So we already we have a (common) example of deliberate use of non-determinism. However, although this reduces traffic to and from the display device (critical on older low-bandwidth channels), it still leaves the considerable expense of updating the map component. Is there any way we can reduce this overhead?

6.6.3 Declarative interfaces

Some years ago, when I was specifying an experimental program editor a slightly different approach to the standard display inertia was taken. As usual, only basic static requirements were specified, and then a few additional requirements were tried that could loosely be described as "cursor inertia". The first of these was strict cursor inertia, where the cursor position on the screen moved as little as possible: this is a dynamic requirement. This had some very strange consequences: in particular, as the cursor began at the top of the document and hence at the top of the screen, it *always* tried to stay at the top of the screen! A second variation on this was always to position the cursor as near the middle of the screen as possible. In terms of observed behaviour this requirement has its own advantages and disadvantages; however, the important thing about it is that it is a purely static requirement. This means that the display map can always be calculated from the current text alone: it is a purely *declarative interface*.

In principle, one would expect a declarative interface to be very predictable. However, few interfaces adopt this style. Harold Thimbleby's novel calculator (ce) does so, but users are often surprised at its features, perhaps because they are unused to the declarative style. The big advantage in performance terms of a declarative interface is that when one only updates the display intermittently, one need only update the *map* intermittently also:



This technique ignores dynamic requirements entirely, so at first glance it appears to be relevant only to totally declarative interfaces. However, given the benefits in terms of performance, we must ask ourselves whether some similar technique could be developed for non-declarative interfaces?

6.6.4 Non-deterministic intermittent update

It is usually the case (and is so in the text editor example) that static consistency is most important semantically. We can imagine then relaxing the demand for dynamic consistency. Normally, it applies between each state transition. However, if we have intermittent update, we could apply it only between displays that actually occur. That is, we effectively bunch a series of commands updating the object and then re-establish the display map based on static consistency, and modify the dynamic requirements to apply to these consistent epochs.

Put in formal terms, previously we would have demanded for each command c_i an object–map pair where each o_i is derived from the previous object via the command, where each $< o_i, m_i >$ is consistent with respect to the static requirements, and where the successive pairs $\{< o_{i-1}, m_{i-1} > , < o_i, m_i >\}$ satisfy the dynamic requirements. Now we ask only for an object for each command, a set of epochs e_j and maps at these epochs only, such that at each epoch $< o_{e_j}, m_{e_j} >$ is consistent and each successive epoch pair $\{< o_{e_{j-1}}, m_{e_{j-1}} > , < o_{e_j}, m_{e_j} >\}$ satisfies the dynamic requirements.

6.6.5 Is it a good idea?

What sort of effect would this have in practise? Imagine the text editor with display inertia. The cursor is on the second to last line of the screen, and we slowly enter DOWN, DOWN, UP; at the second DOWN the screen would scroll to accommodate the new cursor position, then on the UP it would stay (by display inertia) in this new position. If, however, we entered the commands quickly and they were bunched for processing, then after the three commands when the static and dynamic invariants are enforced, the new cursor position is within the original display frame and this is retained – the result is non-deterministic!

Comparing this to intermittent update, we see that in that case only the intermediate states of the screen differed, whereas in this case the final state differs depending on the exact timing of input.

I would argue that the non-determinism introduced is acceptable for two reasons. Firstly, because it is non-deterministic only within the bounds of the static consistency, it is *limited non-determinism*. Secondly, the exact operations of the dynamic requirements are usually unpredictable anyway, and thus the apparent non-determinism will be no worse, and we may actually reduce it. This is especially true of the principle of display inertia. The reason why it is introduced is to ensure that the location of useful information changes as little as possible in order to aid visual navigation. If intermediate displays are not produced then the deterministic strategy leads to a changing display, whereas the non-deterministic strategy leads to a fixed one; clearly, the latter application of the principle is more in line with its intention, and for the user is probably *more* predictable than its application to imaginary intermediate displays.

6.7 Discussion

We have seen how formal non-deterministic models are useful in describing interactive system behaviour. We asked ourselves whether this had any real meaning. In §6.3 we saw that the appropriate user-centred definition of non-determinism was a behavioural one, and under this definition interfaces did display non-deterministic properties. Further, we have seen many examples of how non-determinism arises in practice. In §6.5 we found that users have many ways of dealing with non-determinism, and that the designer can design systems to aid them in this. Finally, we have seen that it can in fact be beneficial to introduce non-determinism deliberately into the user interface, both to achieve other goals (in the example, efficiency) but also potentially to reduce the total non-determinism of the interface.

What have we gained by this analysis? Firstly, by using the strong word non-determinism rather than weaker ones like unpredictability, we see rather more the urgency of the problems considered. Secondly, we have found that many different well recognised problems can be considered as manifestations of non-determinism; this allows the possibility of cross-fertilisation between the domains. Thirdly, by considering problems in this light, it enables us to see more clearly ways of expressing them, for instance, regarding the problem of sharing as that of specifying acceptable levels of non-determinism. Thus we might describe a mail system, not in terms of multiple users and messages, but rather as a single user with a system displaying limited non-determinism. Finally, the analysis has given us a more pragmatic view of non-determinism and hence the ability to consider the idea of deliberately non-deterministic interfaces.