CHAPTER 5

The myth of the infinitely fast machine

5.1 Introduction

Most specification and documentation of interactive systems covers only steady-state functionality. That is, the effect of each user command on the state and display of the system is described, but the effect of lags between the entry of these and when the system actually responds is ignored. Effectively, the system is seen as executing on an infinitely fast machine! The obvious exceptions to this are those systems which deal explicitly with real-time phenomena: response timing, games, simulations, etc. However, it is those systems whose real-time behaviour is incidental which this paper addresses.

There are good reasons for ignoring real-time behaviour. Specifications are deliberately aimed at some aspect of the functionality of a system, and from this abstraction they obtain their power of concise expression. It is therefore important that any considerations of non-steady-state behaviour do not clutter up these abstractions unduly, and can be considered independently. For this reason most of this book has deliberately ignored timing issues. Other formal techniques are equally silent on such issues, for example all the contributors to Harrison and Thimbleby's recent book on formal methods in HCI (Harrison and Thimbleby 1989).

Again, documentation is often complex enough anyway. It is comparatively easy to describe steady-state behaviour in terms of changes in the objects manipulated and snap-shots of screens, but it is both difficult and probably confusing to describe dynamic behaviour without on-line or video presentations. Further, the exact dynamic behaviour may depend on the run-time environment and machine, whereas manuals are often written in a machine-independent fashion.

Both these considerations prompt one to look for ways of dealing with real-time behaviour which are both simple and can be factored out from the main time-independent description of functionality. The next section examines various system compromises that are used to approximate the ideal machine (*buffering* and *intermittent* and *partial update*), and the problems to which these give rise. Section §5.3 introduces a simple formal model which includes temporal behaviour and uses it to give precise descriptions. This provides a framework within which to make precise the problems which arise and the possible solutions. In particular, a precise definition is given of the *steady-state functionality* of a system and of typical system behaviour, such as buffering, as they appear to the user, without regard to a particular model of implementation. This model is then used as a basis for proposing properties that a dynamic system must obey if its steady-state functionality is to be usable. Various concrete expressions of these properties are discussed in §5.5, both existing techniques and novel, including analysis of their user impact. These techniques put demands on the surrounding system software and hardware which are outlined in §5.6, which show where changes are needed in order to put these proposals into practice. Finally, these features are brought together in a design approach where the dynamic aspects of system behaviour are considered separately from the steady-state functionality.

5.2 Compromises and problems in existing systems

In reviewing the research into computer response times, Shneiderman (1984) found that, with a few exceptions, faster is better. This is certainly the general approach taken when implementing interactive systems. At first, care is taken to obtain general efficiency, concentrating on those areas where serious lags are expected. Then, where this proves insufficient, either because the computer is too slow, or because the output device cannot handle the rate of refresh, various additional strategies may be used.

The most common such strategy is for either the application or its surrounding environment (operating system, window manager, etc.) to provide *buffering* to avoid loss of user commands. So normal has buffering become that its absence can be quite unnerving. Slightly less frequently seen are *intermittent update* strategies where the application only occasionally updates the screen: (Stallman 1981) for instance, after receiving 10 scroll-up-line commands the system may decide to display only one screen scrolled up 10 lines, ignoring the intermediate stages. Some versions of the word processor Wordstar augment this by adopting a *partial update* strategy, where the system attempts to keep the current cursor line permanently up to date but updates the rest of the screen only intermittently as it has the time. Both intermittent and partial update are particularly useful when the output device is the limiting factor. They also save some processor time. In Chapter 6, I will suggest *non-deterministic intermittent* update to reduce processor time further, but the analysis presented below does not cover this more complex case.

Not only can dynamic behaviour be confusing for the user, but it can also seriously undermine various desirable properties. A system that could be described as "what you see is what you have got" (cc) on an ideal machine may well not be so on a real buffered machine, as the lag means that the current context as measured by the commands entered so far but still unprocessed, is not the context displayed. A case of "what you see is what you had"!

Problems arise particularly when applications are embedded within a surrounding environment; for instance, multi-processing often leads to long delays which would invalidate a partial update strategy. Further, the environment's strategies may conflict with those of the application, for example the operating system's buffering and the application's buffering may interact badly.

The worst problems occur when the user feedback loop is important, in other words, in the most interactive systems. Key-ahead is usually no problem so long as delays are not too long, but cursor movement and mouse positioning are far more critical. When using cursor keys, delays in cursor movement may lead to entry at the wrong point, and in the worst case to *cursor tracking*. This is the phenomenon where, because of display lag after using cursor keys, the cursor is moved too far and misses its target, then in moving back it again overshoots. Users develop strategies of their own for dealing with this, moving one character in the opposite or an orthogonal direction after a long sequence of moves, or even inserting a character just to see if the cursor has stopped. These strategies become so automatic that some users claim that cursor tracking is not a problem. On the other hand, for small moves, feedback is not so critical: correcting typing mistakes on the fly is relatively easy since the number of moves to make is known, and can be an almost automatic reaction.

Mouse-based systems pose more serious problems. Some systems obtain the mouse position independently of button clicks, and thus, for instance, in a drawing application, depressing a button and moving will result in a start point somewhere along the mouse's path. Users of such systems get used to a "click and hold" strategy when drawing or manipulating icons and menus. Because the meaning of a mouse action is so heavily dependent on the display context, some systems disallow mouse-ahead. Again this leads to a "click and wait for confirmation" strategy for the user. The task of accurately positioning the mouse cursor already places high cognitive demands on the user, which are increased if the user must be constantly vigilant to check that the various mouse actions have had the intended effect. This is stressful, makes it difficult for the user to think ahead, and prevents the user developing effective motor responses. In particular, the illusion that the mouse is an extension of the user (Runciman and Thimbleby 1986) is destroyed.

5.3 Modelling

In this section we develop an abstract formal model of interactive systems, taking into account the timing of input and output. We will use this to define explicitly what is meant by steady-state functionality, to describe the various departures from this ideal functionality and the various observability constraints that are required to make such a system usable. In line with the philosophy of this book, the model will as far as possible be a surface model, in the sense that it will try to describe the system from the outside. For example, the definition of a perfectly buffered system is not in terms of a physical representation, such as a block of memory in which keystrokes are stored, but in terms of the behaviour of the system. This approach both avoids odd hidden problems (e.g. the system may save keystrokes, then later ignore them) and, by staying outside the system, is a description of its interface only and has therefore a greater validity for HCI.

5.3.1 A simple temporal model

There are many possible formulations, and the one presented here is chosen because it is most similar to the previous models. We consider a sequence of discrete time intervals. At each time interval the user may enter a single command (from a set C) or do nothing (represented by τ – a tick). Also, with each time interval, we associate a current display (from a set D); however, we assume that this occurs just after the input for that interval, so that there is time for a fast computer response, but so that the gap is imperceptible to the user. Thus the display in the same interval can be seen as the instant response, but lags are represented by responses several time-steps later.

The display is obtained from the current machine state (from a set E), by a function $display : E \to D$. This state comprises all the computer's memory of the past. The relation of this state to the input history can be represented in two ways, either by considering the whole history at once, or by an incremental approach. The first method calls for an interpretation function (I) which gives the state due to any input history. The input history (from a set H) will consist of sequences of commands (from C) and ticks (τ):

$$I : H \to E$$

The alternative is to use a state transition function, which, given the input for any time step and the last state, gives the new current state:

$$doit : \; (C \cup \{ \tau \}) \times E \rightarrow E$$

These two representations are of course not independent and are linked by the equations:

$$\begin{array}{lll} I(\,[\,]\,) & = & e_{init} \qquad \text{– the initial state} \\ I(\,h :: c\,) & = & doit(\,c, I(\,h\,)\,) \end{array}$$

That is the state obtained for a history is obtained by starting at the initial state and repeatedly applying *doit*. We could call this a temporal PIE or a τ-PIE.

Note that the results of Chapter 2 concerning monotone closure mean we do not have to worry about the full abstractness of assuming a state. In fact, we could proceed without the concept but the definitions would become more complex.

5.3.2 Defining steady-state functionality

To talk about steady-state functionality, we need an idea of when the system has stopped changing. This is clearly when further time intervals with no user input (i.e. τs) don't alter the state:

$$e \text{ stable} \quad \equiv \quad doit(\,\tau, e\,) = e$$

If the state does not change, then the display, which is obtained from the state, clearly doesn't change either. However, just because the display is stable, it does not follow that the state is. For instance, in a multi-processing environment, a background task may well be altering the file system but not produce any visual effect, or, when a large global edit is being performed, there may not be any indication of completion; in the most extreme case, consider a user response timer continually ticking away with no visible change until the user responds and the delay is recorded.

We will define steady-state functionality by associating it with the response obtained by a patient user who always waits for the system to stabilise before entering more commands. Unfortunately, from the last example, we see that there are in fact systems that never stabilise: another similar example would be an alarm clock. However, it is usually the case that either the system does always stabilise, or that the parts that do not, can be abstracted away. Making this assumption, we can associate with any sequence of inputs a longer sequence with additional ticks on the end, which gives rise to a stable state. These ticks can be thought of as the time a patient user would have to wait:

$$steady : H \rightarrow H$$
$$steady(\ h\ ) = h :: w$$

where w is the shortest sequence of τs such that $I(\ h :: w\ )$ is stable.

If the machine were infinitely fast, it would reach a stable state instantly, and w would be empty. As there would be no time to type-ahead, all users would effectively be patient. For any sequence from H, we can obtain the sequence that a patient user would have entered:

$$patient : H \rightarrow H$$

$$\begin{array}{lcll} patient(\ [\,]\ ) & = & steady(\ [\,]\ ) & \\ patient(\ h :: \tau\ ) & = & patient(\ h\ ) & \\ patient(\ h :: c\ ) & = & steady(\ patient(\ h\ ) :: c\ ) & \text{when } c \neq \tau \end{array}$$

We can now define the steady-state functionality to be exactly that which the patient user would obtain: that is, I composed with *patient*. This is precisely the functionality defined by most specifications and described in most documentation, and is therefore the behaviour that should be validated against them.

5.3.3 Buffering and update strategies

How does this differ from the actual functionality observed by a less patient user? To see this, we compare the steady state obtained from a sequence of commands (and ticks) h with that obtained if the user had been patient. That is:

$$I(\ steady(\ h\ )\ ) \quad \text{compared to} \quad I(\ patient(\ h\ )\ )$$

If these are equal, then the system obtains the same steady state no matter how fast the user enters commands. That is, equality in the above is the definition of a *perfectly buffered system*:

perfect buffering:
$$I(\ steady(\ h\ )\ ) = I(\ patient(\ h\ )\ )$$

Note that this definition is purely in terms of the behaviour of the system; it makes no reference to a buffer as an implementation device. Later on we shall consider departures from perfect buffering.

We can also examine total versus intermittent display update using this scheme. We say a system has a total update strategy if all steady-state displays occur in the actual display sequence:

total update:

$$\begin{array}{l} \forall\ h \in H,\ \ h_0 \le h_1 \le h_2 \cdots \le h_n \le h \\ \quad \exists\ \ s_0 \le s_1 \le s_2 \cdots \le s_n \le \mathit{steady}(\ h\ ) \\ \qquad \textbf{st}\ \ \forall\ i \quad \mathit{display}(\ \mathit{steady}(\ h_i\ )\ ) \ = \ \mathit{display}(\ s_i\ ) \end{array}$$

A large proportion of systems are perfectly buffered and adopt a total update strategy: this is because they are effectively programmed for an ideal machine and buffering is supplied by the environment. The environment can supply degrees of automatic intermittent update in addition to buffering; for instance, TIP, (ce) a screen control package, attempts to suppress display update when there is input pending.

The decision whether or not to update is not just one of efficiency. Some intermediate displays convey information by their dynamic behaviour: for instance, seeing a display scroll up line by line gives the user more of an impression of direction than a sudden jump. On the other hand, if the user has typed ahead it is likely that any such feedback will be too late!

5.4 Dealing with display lag

As we have said in §5.3, the interpretation that the user gives to commands depends on the display context. The problem of display lag can be addressed in two ways:

- A computer response, trying to reinterpret or ignore commands.
- A user-based solution, supplying sufficient information for the user to make sensible decisions.

Typically a system will use a mixture of the two; in fact, we will see that responses of the first kind necessitate the provision of specific information for the user.

5.4.1 Computer response

In the first category, the most obvious example is disallowing any mouse-ahead. This amounts to:

$$\begin{array}{l} h \ne \mathit{steady}(\ h\ )\ \ \textbf{and}\ \ m \in \mathit{mouse\ commands} \\ \qquad\qquad \Rightarrow \quad I(\ h :: m\ ) \ = \ I(\ h :: \tau\ ) \end{array}$$

This says that when we are not in steady state, any mouse command is treated just like a tick, an interval with no input. That is, mouse commands are ignored out of steady state. This may be too rigid, however, and mouse-ahead may well be acceptable if the displayed context when a command is issued is the same as the context when it is eventually applied. For instance, if there are two independent editing windows, mouse-ahead in one would be allowable even if there were commands outstanding for the other. Even if the contexts are different, the denotation in the context of invocation may still have meaning at the time of application. For instance, if there are still keyboard entries outstanding for a find/replace buffer and the user wants to initiate the action, then, if a simple function key was being used, we would expect most applications to queue it; it seems reasonable therefore that the lexical change to a find/replace button icon should behave similarly, allowing mouse-ahead.

A system that disallows some commands, but not others, can be represented by the existence of a function *noted*:

$$\begin{array}{l} \mathit{noted} : H \rightarrow H \\ \forall\, h \in H \quad \textbf{let} \quad n = \mathit{noted}(h) \\ \qquad n \text{ is a subsequence of } h \\ \qquad \textbf{and} \quad I(\mathit{steady}(h)) = I(\mathit{patient}(n)) \end{array}$$

This says that for any history, we can associate a shorter history of commands that have been taken note of. These noted commands are a subsequence of the original commands (dropping out the ignored ones), and the behaviour of the system is the same as if only the noted commands had been patiently entered. Note the similarity between this and the definition of perfect buffering in §5.3.3. This similarity is because systems that possess such a *noted* function will obey the exception rule "no guesses" (Chapter 2 §2.9), which suggests that when exceptions occur a system should not perform a special action, but instead merely ignore the command that raised the exception.

5.4.2 User solutions

The user should be able to discern what the correct context is. In particular, is this steady state? This decision must be possible from the contents of the current display, requiring the existence of a decision procedure for the user, *is_steady?*:

$$\begin{array}{l} \mathit{is\_steady?} : D \rightarrow \mathit{Bool} \\ \textbf{st} \quad \forall\, e \in E : \ \mathit{is\_steady?}(\mathit{display}(e)) = \mathit{stable}(e) \end{array}$$

Although cast as a user decision function, its importance is of course the information it requires of the system. This information could be positive (e.g. a status light when the system is stable), or negative (a red cursor when it's not): formally both give equivalent information. Note, however, that the system response producing the information must be immediate: it is no use waiting for steady state for information to say the system wasn't in steady state when you typed the command! This is why the definition refers to the current state and its display. Some systems are always in steady state except when the screen is actually changing (e.g. cursor moving or text scrolling), and thus lack of change indicates steady state. We could modify the above condition to look at several displays; however, it is reasonable to maintain the definition and regard change as being an attribute of a single timeframe.

The user needs to determine whether the current context is steady, both to make valid decisions based on the display, and to predict dynamic response for cases like mouse-ahead. In the latter situation we see that there is interference between the computer's strategies for dealing with dynamic behaviour and the user's. For instance, if a user changes windows, the window manager might throw away key-ahead until the window has been displayed, to avoid accidental typing to the wrong window. This could be very annoying if the user does not know when type-ahead can begin again.

If the simple strategy of no mouse-ahead is used then the existence of *is_steady*? is sufficient. If, on the other hand, one of the context-sensitive mouse-ahead strategies is used a more complex observability criterion must be introduced. This will be related to the function *noted* introduced above. So that the user can know whether a command has been noted or not, we need a user predicate similar to *is_steady*?. We will call this *is_noted*?:

$$\begin{aligned}&\mathit{is\_noted?} : D \rightarrow \mathit{Bool}\\ &\textbf{st}\ \ \forall\, h :\ \ \mathit{is\_noted?}(\ \mathit{display}(\ I(\ h :: c\ )\ )\ )\\ &\qquad\qquad \Leftrightarrow\ \ \mathit{noted}(\ h :: c\ ) = \mathit{noted}(\ h\ ) :: c\end{aligned}$$

That is, *is_noted*? tells us immediately whether or not our command will have an effect in steady state. An example of this would be a buzzer sounding whenever a mouse-click has been ignored. Perhaps even better would be to predict beforehand:

$$\begin{aligned}&\mathit{will\_be\_noted?} : D \times C \rightarrow \mathit{Bool}\\ &\textbf{st}\ \ \forall\, h :\ \ \mathit{will\_be\_noted?}(\ \mathit{display}(\ I(\ h\ )\ ),\ c\ )\\ &\qquad\qquad \Leftrightarrow\ \ \mathit{noted}(\ h :: c\ ) = \mathit{noted}(\ h\ ) :: c\end{aligned}$$

That is, *will_be_noted*? tells us before we enter a command whether or not that command will have an effect. An example of this would be hour-glass mouse icons. (Goldberg 1984)

If the user is able to tell from the display whether or not a command has taken effect (but not of course whether the effect has been as expected, which requires steady state), then certainly the system needs (in some sense) to know. This disallows retroactive decisions, such as circular overwriting keyboard buffers (as in some UNIX systems). In practice this will mean being conservative about allowing mouse-ahead, only accepting it if we can be sure that the current context and the steady state context are in the right relation.

Note also that the information for these decision functions must be available *all* the time. In particular, it cannot be subject to intermittent or partial display strategies.

5.4.3 Summary of formal analysis

By considering the problem of real-time behaviour formally we have:

- Related precisely the steady-state functionality described in most specifications and documentation, to the actual temporal behaviour experienced by the user.
- Shown how the departures from this steady-state functionality can be described and recorded.
- Used the formal description of these departures to expose specific information that must be available to the user: *is_steady*?, *is_noted*? and *will_be_noted*?.

5.5 What would such systems look like?

Supplying sufficient information to ensure the existence of such decision procedures is not sufficient; the information must also be in a form comprehensible and noticeable to the user. We should also take account of whether we expect the user to be acting consciously or automatically, and where we expect the centre of attention to be. For example, a user having just asked for a program to be compiled would expect it to take a while, and would consciously look out for indications of completion (e.g. a new prompt); on the other hand, if the user were typing fast into a word processor, we could not expect the same degree of awareness. Again, if mouse-ahead is prohibited, flashing the mouse cursor would be an acceptable signal since attention is likely to be centred on the mouse, yet signifying a locked keyboard by a flashing LED on a key would be unacceptable for touch typists. Monk (1986) discussed this and similar issues when considering signalling modes to the user. In fact, the various conditions considered here (the existence of type-ahead and ignoring commands) could be thought of as a form of modiness.

We look first at information to represent the steady-state. Here there is a well-established use of hour-glass, busy bee or watch icons, substituted for the normal mouse icon, when the computer is busy. However, these tend to be used only when some significant pause, such as file transfer or global editing, is occurring. When used they tend to signify that either mouse input or possibly all input is being ignored. Thus they fall closer to the *will_be_noted*? than the *is_steady*? information. However, on such systems, it is often the case that mouse clicks can be ignored in other contexts also. The mouse icon could be used consistently in all non-steady-state situations, especially if we are using the rule of ignoring all such mouse input. This would be acceptable only if we could be sure that in all critical situations attention is focused on the mouse cursor, which does fortunately seem quite likely. The alternative centre of attention is the text entry point or current selection (the only one in non-mouse systems). Similar mechanisms can be used here; however, on more conventional terminals it is unlikely that there is much choice of cursor attributes, although the one choice that is often available is between flashing and steady cursor, and this could be quite appropriate.

The disadvantage of making steady-state information so prominent is that it could be a significant distraction: imagine the cursor changing icon for a fraction of a second on each keystroke! If we expect most uses of steady-state information (as opposed to explicit *will_be_noted*? information) to be in situations where users expect problems, then it could be consigned to a status line, either as a simple flag or more elaborately. Perhaps the most extreme form of such information would be the *munchman buffer*. As you type, the characters appear on the bottom line of the screen. As the application deals with each character, it is consumed in the bottom left corner (by the munchman!). Although slightly strange, such an interface might be useful in certain circumstances, such as command interfaces. Perhaps more likely, particularly where quite powerful personal computers are used as terminals, would be to have an icon associated with a window representing the state of the interaction. In its collapsed form it could have a small bar indicating the fullness of the buffer. When expanded, more information could be available; in particular, this could be combined with the status information for a terminal emulator. This would again be of particular value with command-based interfaces: it allows the user to determine steady-state, to make predictions about the context where commands will be interpreted, and gives feedback as to progress which can be very important if response is slow. (Shneiderman 1984)

If we consider mouse-ahead acceptability, or keyboard locking, then changing mouse or cursor attributes again becomes by far the best solution. For the mouse these attributes will be context sensitive, so that the mouse may display a locked form when over a text window with outstanding updates, but be in its normal state when moved over a control panel where mouse input is acceptable. This effectively supplies us with the *will_be_noted*? information. However, even though it may be safe to assume the user's attention is focused on the mouse or cursor, merely visual cues may not be noticed if the user is acting semi-automatically. Thus it is insufficient to supply only *will_be_noted*? information: feedback (preferably aural) is required when the user makes a mistake. That is, we need to have the *is_noted*? information also, and in a different medium. The usual objections to such feedback (people don't like machines that beep at them) don't hold here, as there is sufficient visual warning before the fact: the self-aware user can use the visual *will_be_noted*? information and thus need never hear a thing.

5.6 System requirements

If any of the strategies proposed or any alternatives are adopted, and are only partly correct, then they may be worse than useless. Users may be lulled into a false sense of security and then make mistakes which would go unnoticed because of the reduced attention, or they may eventually come to ignore the feedback, considering it an unreliable and therefore irrelevant distraction.

Successful implementation depends on the mutual cooperation of the application, the environment (operating system, window manager, etc.) and the interface peripherals. We will see that one recalcitrant partner can spoil the system. Any of these parties, or a combination of them, can take the initiative in the process. Thus we will consider each of these (environment, application and peripheral) in turn.

5.6.1 Environment control

Except where communication is over large, slow networks, we can assume that the operating system has almost immediate control over its interface hardware. It may not be able to refresh an entire screen in an imperceptible interval, but at least it will be able to guarantee immediate control over some portion of the screen, enough to display some status information. Several problems may arise, however, if the application does not cooperate.

If the application buffers input for itself (e.g. in multi-user systems where system calls are expensive), the environment cannot know how much has been used and hence can give only coarse estimates to the user (i.e. no per character munchman buffers). In single-user systems (for instance CPM), system calls are usually on a per character basis and the system can thus keep a more exact track of input pending. In both cases if the operating system allows polling of input then there is the possibility of busy waits, not allowing the environment to detect steady-state. This will happen, in particular, when the application is attempting to perform screen update optimisations, as it will poll to decide whether or not to update: this is a case of antagonistic temporal strategies. Clearly the application must pass some message back, for instance guaranteeing to wait on input eventually. More problems arise when considering non-noted events. Unless there is a system-wide policy, such as ignoring *all* mouse-ahead *always*, only the application can know the allowable contexts. This is alleviated if the operating system includes a rudimentary user interface management system. (Pfaff 1985) The actual application would be perfectly buffered, and a separate lexical/syntactic module would determine acceptable events, this second module guaranteeing to run fast enough to be effectively instantaneous.

5.6.2 Application control

There are several reasons for investing temporal control in the application. It has most knowledge about the way it will be used and what sort of displays and feedback are appropriate and consistent with the rest of its operations. Further, the turnover of applications is far greater than that of operating systems and thus new ideas can be included more quickly. Finally, and more tentatively, similar to the desire for consistency across a system, is the desire for consistency of the same application on different systems.

The application is frequently prevented from exercising control by the primitives supplied by the environment. On the input side, only blocking input may be supplied, not allowing the application to decide whether there is any pending input. This is true, for example, of older Unix systems. In these cases intermittent update strategies are often possible by requesting large chunks of input and updating on each chunk; however, steady-state indicators are very difficult, even when scheduling can be guaranteed to be frequent.

Scheduling problems combine with the paucity of primitives to prevent proper synchronisation of input and output. Imagine we are about to output a screen that comprises a major context shift, and we wish to ignore all outstanding input. The operating system provides a *flush_os_buffers* call that does immediately flush all of its outstanding events. The following sequence is no good:

```
flush_os_buffers();
update( screen );
```

because the application may be descheduled between the two lines and further input may be entered before the screen is displayed. Even if there is no delay, the update itself may take some time and unwanted input may still be accepted. However, if we reverse the order the situation is worse:

```
update( screen );
flush_os_buffers() ;
```

If there is descheduling, the user's input will be ignored, despite the fact that a new screen is displayed which, among other things, will imply that input is acceptable. On the other hand, if there is no delay, the call to *update* is likely to mean that the request has been placed in the appropriate queue, but not that it has completed, thus leading to the same problems as the previous order. These problems are particularly evident with bit-map displays where the refresh time can be significant.

Where events are time-stamped, we might try to compare these time-stamps against the time of last update; however, obtaining this time leads to the same problems as before. It is only the front-end software that can meaningfully synchronise input and output, and thus this must supply the relevant information.

The simplest solution therefore, which could be provided at little cost, is to have the window manager or tty-driver, upon request, insert an event into the input stream when an update completes. The application would then know that any events received before this were entered while the screen was not up to date.

5.6.3 Peripheral control

Poorly designed interface hardware can destroy attempts at good temporal behaviour, perhaps by having uncontrollable buffering, or insufficiently rich operations; however, in practice this is usually an annoyance which can be overcome. On the other hand, the peripherals rarely contribute anything positively. This need not be the case. Terminals are getting increasingly intelligent and this intelligence can be used. To a large extent, a terminal taking control has the same problems with the underlying system as the operating system has with the application; however, there are cases where it could be worthwhile despite this. For instance, where a PC is acting as a terminal over slow lines or a network, it may well be worth reporting the state of its buffers even when there may be additional buffering in the host. Perhaps, more importantly, the terminal can cooperate either with the operating system to take workload from it, or with the application despite a possibly antagonistic operating system. This latter case is most interesting, as it can overcome the technical inertia of the operating system. For instance, synchronisation, such as suggested above, where output events generate input tokens, can be accomplished trivially (in fact some VDUs have a who-are-you request which can be used for this purpose). Instant feedback is more difficult, as this has to be handled by the terminal if fast scheduling cannot be ensured. However, the solutions proposed for environment control apply here. Either catch-all policies such as no mouse-ahead, or simple descriptive mechanisms such as region-sensitive mouse-ahead, could be provided by appropriate control codes. The use of PCs as terminals, specially designed programmable terminals such as TIN (Macfarlane and Thimbleby 1986). or intelligent workstations such as the BLIT, (Pike 1984) allow the possibility of downloading programs of significant complexity which could in particular handle the parts of an interface requiring close synchronisation and rapid response, leaving a bare-bones application in the host.

5.7 Conclusions – a design approach

The above discussion shows that it is possible to talk about and build fully temporal systems, by embedding a steady-state design in a real temporal system. From the above discussion, we have the beginnings of a design approach:

(i) Take a steady-state functionality.

(ii) Decide under what circumstances to allow partial or intermittent update.

(iii) Decide in what circumstances to ignore user commands.

(iv) Choose ways, not subject to partial or intermittent update, to represent the achievement of steady-state, to signify when user input is acceptable and to give feedback when input is ignored.

Stages (ii–iv) are considered separately from the first. In particular, they would be specified and probably documented separately, there should be support for them from packages, the environment, the peripherals or some combination, and further there may be system-wide policies for them, especially for presentation (stage iv).

Following such a design approach should lead to systems with real-time properties that are easier to specify, easier to document and easier to use.