CHAPTER 10

Events and status – mice and multiple users

10.1 Introduction

If we look back over the models presented in the previous chapters, we find an imbalance between the interactions from the viewpoints of the computer and the user. In the PIE model, the user entered a sequence of commands, and the computer responded with a sequence of displays. At first glance, the formal models look fairly symmetric between input and output. This impression changes if we examine the domains of these sequences and their interpretation.

One obvious difference between these two sequences is the small number of possible user commands compared with the vast number of possible displays. That is, there is a mismatch in bandwidth between the user's inputs and the computer's responses. Although this feature will not be central to this chapter, it is an important issue in its own right. The mismatch is reasonable, since it is quite possible for computer systems to produce displays at a rate of several million pixels per second whereas typing at such a rate would be hard. Of course, we will not be able to take in information at that huge rate, but even so, our ability to gather information from a display is still far greater than our rate of generation. If we want to allow the user to control the system (rather than *vice versa*) we need to design the dialogue carefully so that the computer's high output bandwidth can help users increase their effective input bandwidth.

The second difference is one of persistence. Considering the red-PIE model, although it said nothing about the precise temporal behaviour of the displays and commands, there were implicit assumptions in the definition of properties. For instance, the simplest form of predictability was the ability to tell where you are from the current display. The scenario was: I interact with an application, go away for a cup of tea (a not infrequent occurrence), then upon returning want to

know where I've got to. If I use the current display to obtain this information, it can only be the persistent part. Any beeps or whistles, although very important to signal errors whilst two-finger or copy typing, would be useless as status indicators upon my return. Other forms of predictabilty involve exploratory activity by the user and thus assume the user's presence. For these forms of predictability, persistence is not so important. Paradoxically, such exploratory behaviour does not typically encounter non-persistent parts of an interface.

To distinguish these levels of persistence, we shall talk about *status* and *events*. Status refers to things which always have a value, such as the current display or the temperature. Events are things that happen at a particular time, for instance user's key strokes or the computer's beeps. In fact, the distinction is nothing like as clear-cut: events are rarely instantaneous, and may often be regarded as rapidly changing status. We briefly examine this interplay between the two in the next section.

Succeeding sections look at how to include status inputs such as mouse positioning within formal models. We can use these models to classify the way systems respond to status inputs. In particular, we find there is a close correspondence between the complexity of the models required to describe a particular system and the hand–eye coordination required.

Finally, we consider how the distinctions between status and events can help in the understanding of multi-user systems such as mail or conferencing.

10.2 Events and status – informal analysis

As we have already noted there is an interplay between status and events which clouds the immediate distinction. The way we classify a particular phenomenon will be partly subjective, and in particular depends on the temporal granularity with which we view it.

Consider for example a mechanical alarm clock. If we ignore its inputs (i.e winding it up, and setting times) and concentrate on its outputs, we have two distinct channels: its face, which displays the current time, and its bells, which wake us up in the morning. This seems like a fairly clear example of status and event. The time displayed varies continuously and is always available, while the alarm rings at a particular hour of the morning.

Let us look in more detail at the alarm's bell. If I happen to be rather sleepy, the clock may ring for a while. That is, it becomes a status output. The important events then take place when it starts ringing and when I eventually turn it off. So, if we look at the clock with the granularity of hours in a day, the ringing of the bell is an event, but if we look at it with a granularity of seconds (or minutes!), whilst I am actually waking it is a status.

The clock face is even more complex. If, say, the alarm is not set, but I want to do something at five o'clock. I keep looking at the clock, and when the hands pass the hour I react. I have interpreted a change in status (through the five o'clock mark) to be an event. This turns out to be quite a general phenomenon: any change in status can be regarded as an event. Notice also that in order for the change in status to become an event, I had to watch the clock. Probably I would do this periodically, so there would be a slight gap between the objective event of the clock passing through five o'clock and the subjective event of my noticing it. Again, these are general phenomena. Frequently, some procedure is carried out to monitor a status indicator without which changes in status could not become events. Further, such procedures will typically induce lags between the events noted by different observers.

Now, if we look closely at the clock face, we notice that the second hand moves rather jerkily. Inside the clock, several times a second, the spring vibrates and moves a ratchet which allows the clock gears to move on one notch. This discrete movement is visible on the second hand, and present but rather hard to see on the minute and hour hands. In fact, if we look closely at a large grandfather clock the jerky movement of the minute hand becomes visible, and the hour hand probably jerks noticeably on Big Ben. That is, at a fine grain, the change in status of the clock face is due to events – the ticks. Of course, we could drop another level again and describe the position of the cogs and springs to yield a status description of the entire workings of the clock.

So, even on something as everyday as an alarm clock, we see quite a subtle interplay between status and event descriptions. As most digital computers operate in discrete time steps it would be possible to describe them completely in terms of the events specifying the changes at each instant. This is very obvious if we consider a terminal attached to a time-shared system. The changes to the terminal's screen are sent character by character down a communications line. In principle, if users had access to this event information they could reconstruct what their screens should look like.

From a formal point of view, we could model the display in this way. User input events give rise to system output events which specify the way the screen is to change. In fact, this is precisely the view programmers are usually given of the interface. Even apparently continuous graphics displays are usually the result of a stream of change requests from the program. This does not describe the way the user sees the system. Most of the changes happen at timescales well below the user's limits of perception, and even when they do not they are often intended to (shades of the infinitely fast display). So, most of the time it is right to view the display as status, and any model that treats it as a sequence of change events will be inadequate for understanding user behaviour.

On the other hand, just as the movement of the clock hands through five o'clock was interpreted as an event, the user may interpret certain changes in the status of the display as events. So, the user may not regard the individual characters which are drawn on the screen as events (although the programmer might). However, if all these changes together mean that a box has appeared in the middle of the screen with the words "fatal disk error", this is a significant event. Note that even if the change in status has happened in a very "un-eventy" way for the program, perhaps by changes to a memory-mapped screen, the user is still free to interpret the change in status as an event.

The granularity with which we choose to classify events and status depends in part upon the tasks which we are considering. So, for the alarm clock, if the task were getting to work on time then the ringing of the alarm clock would be an event. If, on the other hand, the task were the act of waking and getting out of bed, a status interpretation would be suitable. Thus if we wish to inform the user of an event, we must think about the pace of the task to which it contributes. For highly interactive rapidly paced tasks we would need events that would be recognisable at fine granularity: a buzzer, for example. If the task is long lived and low paced, such as the building of a tower block (or cottage), then an event, say the completion of a phase, would not warrant such an intrusion; a memo would probably suffice.

So, the distinction between event and status depends on granularity and interpretation. For the user, it also depends on salience. The appearance of a message at the top of my screen may be viewed as a status change event objectively, but it has little value unless I notice it. An event for *me* is something that I notice and act upon.

10.3 Examining existing models

We can now examine the models we have already considered in the light of the status/event distinction. This distinction can also shed light on other dialogue specification formalisms.

We started the chapter by noting that the PIE model was basically an event-in status-out model. Most of the other models follow in a fairly similar vein and to a large extent share this property. The exception is the complementary view model of Chapter 9, so let us start there.

10.3.1 Updating views – status-in status-out

In Chapter 9 we considered the paradigm whereby the user manipulates a view of the system. Normally we expect that, as the underlying state of the system changes, then the view changes along with it. The view-update paradigm suggests that, in addition, we allow the user to alter the view and have the system maintain a state consistent with that view. The precise manner by which the user interacts with the view is not part of this model, and may involve event- or status-like inputs and outputs. However, at the level at which it describes a system, this model comes pretty close to a pure status-in status-out model. What is more, both the user's status input and the user's status output are the same thing, i.e. the view.

Obviously, in this paradigm we can think of the changes of view as events, but the way that the model emphasises the static relationship between view and state encourages a status-like interpretation. This is further emphasised if we also recall the condition of *process independence*. This condition said that the changes in the underlying state brought about by a change in the view were independent of the way in which we performed those changes, and were a function only of the initial state and the current view. This form of history independence is not essential for a status-based system. There is no reason why the current state should not be based on the entire history of user status input. However, it does very strongly encourage such a viewpoint.

10.3.2 Status outputs and event outputs

Returning now to the PIE model and its derivatives, we first consider the output domains. As we have noted, several of the properties suggest that the display component is status-like. It is possible to add event-like components to the existing model. For instance, we could define the display of a word processor to be $Bell \times Screen$, where *Screen* is the screen that is displayed and *Bell* is a flag indicating whether the bell rings as the screen is updated. This display domain could be used to describe the behaviour of the system after an error, with the *Bell* component alerting the user to the problem. In fact, the possible error behaviour rules we considered both in Chapter 2 and when considering temporal behaviour assumed that some such component exists. If we have such a display definition, we have to be very careful when applying predictability rules. Some would require that we limit the display considered in the rule to the status part only. In particular, if we want the user, upon returning to the system, to be able to infer anything from the current display, then the display considered is without any event part.

If we consider more complex event behaviour, the system may produce events at instants not directly in response to user events. This could, of course, be captured in the temporal model of Chapter 5. However, the rules developed in

that chapter were aimed at status output systems and would require modification to encompass event outputs. In particular, the main emphasis was on the steady-state behaviour, that is, the display after the system has settled down. This assumes that transient behaviour, including event responses, is relatively unimportant. For example, consider a spoken interface to a directory enquiries system. If the user prods the system with a speech input, the system then responds with speech. The steady state would consider the spoken response as part of the transient, unimportant behaviour. Thus the steady-state functionality, which is supposed to capture the essence of the system, would be... silence. This sounds rather like Beckett!

So, with a little stretching, we can begin to apply our models to event-based outputs. However, the above discussion does warn us that properties that are appropriate for status-like outputs are not necessarily appropriate for events and *vice versa*. It seems wise therefore to distinguish the event and status outputs of a system within a formal model, even if they appear otherwise identical within the model. This is rather like the way we distinguished the display and the result components in the red-PIE model and its derivatives. Looking at the model alone, the two differed only in their name. However, they differed greatly in the interpretation we laid on them and in the way they were used in framing properties.

10.3.3 Status inputs

As one would expect, many of the same warnings apply when we consider status inputs. The most common form of status input is a mouse position, or other form of pointer, but if we took as an example a computer-assisted car, we would also need to consider pedals, steering wheel, etc. Bill Buxton (cc) suggests that computer interfaces could make use of many more different input devices, many of which would be continuous status devices. We will consider properties for status inputs in the next section, so now we will just look at the way they can be included within our existing frameworks.

The PIE model is very heavily entrenched in an event input world. Even within its restrictions, we can describe status input to some extent, but not very convincingly. In the last section we considered the way the screen was updated by many events. In a similar way we can convert the user's status input into a series of events representing changes. For instance, if we were building a red-PIE model of a mouse-based system, we might have the command set as *Keys* + *MouseButtons* + *Moves*. *Keys* and *MouseButtons* would represent the events of hitting the keyboard or clicking a mouse button, respectively. *Moves* would capture the mouse movement with commands like *MouseUp*, *MouseDown*, etc. This is precisely the way many window managers and graphics toolkits deal with the mouse.

These mouse movement commands "work" in the sense that we could write down interpretation functions and display and result maps which would mirror the behaviour of a system. They do not, however, capture the essence of the system for the user. When I move a mouse I do not feel that I am entering a sequence of up, down, left and right commands. What I am doing is moving a pointer on the screen. Arguably, this is true even for text cursors with cursor keys. Much of the routine cursor movement is proceduralised: the discrete commands become unconscious and the effect for a skilled user is a smooth movement of the cursor about the screen. Again, the level of analysis is all-important.

One of the intentions in the study of dynamic pointers was to help deal with mouse-based systems. Do they help us here? By talking about the relationship between positions on the display and locations in the underlying objects they give us a vocabulary for relating certain types of status inputs to status outputs. However, they do not address the specific distinctions between status inputs and event inputs. Where we considered the form of manipulation in detail we assumed that the mouse position was interpreted in conjunction with some event-based command. In fact, we will see that this represents an important subclass of status input systems.

A more generalised view of status input can be derived by considering the temporal model, the τ-PIE. If we add for each time interval a value for the user's status input, we allow system descriptions which include many types of behaviour, dependent on the precise timings of the user's status. Whether the full generality of such responses is a good thing, will be considered in the next section. For now, let us note that by adding status to each time step, there are no pure τ or tick time intervals. Considering event outputs required a radical re-evaluation of desirable properties. Adding status inputs requires a similar re-evaluation and, in addition, leaves some of our constructions in need of a complete overhaul.

10.4 Other models

We now take a quick look at some other dialogue formalisms, and some of the issues they highlight. Several formalisms make use of process algebras, such as CSP. Alexander's SPI, (Alexander 1987b) which has already been mentioned, is an example of this. By the nature of the underlying formalism such descriptions are event-in event-out. Status information is typically conveyed by change events in such descriptions. Most of the formalisms which I classified as of "psychological" origin tend also to regard the user's interaction in an event-driven way. Typically, there is little if any description of the system's response either as status or event, the emphasis being on the user's tasks or goals and their

relation to the user's event inputs.

General computing specification formalisms, presumably because of the discrete nature of most computing, tend to deal only with event inputs. For example, specifications of interactive systems in Z, (Sufrin 1982, cd) and in algebraic formalisms (Ehrig and Mahr 1985), effectively map user events to individual schema or functions which then act as state transformers. The outputs tend to be functions of the state, a constantly available status. This situation is not inherent in these formalisms, which are quite capable of describing status inputs and event outputs; it is just that the event-in status-out description is often easiest.

Sufrin and He's model of interactive processes, (Sufrin and He 1989) which is defined in Z, blurs somewhat the event/status position of its outputs. It inherits the display/result distinction (but calls them view and result) and, like the red-PIE model, describes these as functions on the state. However, unlike the PIE model, these functions are not continuously available, but are defined only for certain states. Now, as the display is clearly not meant to black out during the intermediate states, I take the unavailable states to represent the constancy of the display, and the available states to represent the display *changes*. In fact, they describe the availability of the display and result as "just after" the state change event. That is, they are very close in nature to status change events. On the other hand, Sufrin and He define properties similar to predictability properties, which suggests that at least the display is of a status nature. Also, all the examples used by Sufrin and He have a display for each user command; the non-display states seem to be there to allow for internal events and state changes.

Both the Sufrin and He model and the process algebras lead us to consider whether it is better to deal with the display and the result as event or status.

10.4.1 Display – status or event?

For the display, we have already dealt with something very similar to the Sufrin and He model when we considered alternative definitions of the PIE using functions from command history to effect history. The case where the effect history does not "grow" with the command history is precisely the same as the unavailable states here.

As we have noted, the distinction between status and event is partly subjective. A visual display is "always there" and so my preference is to regard it as status. I would prefer to reserve event outputs for more obvious things such as auditory feedback. There are in-between cases, such as a flash of the screen (sometimes used as a silent bell) or the *appearance* of a dialogue box which marks an exceptional behaviour.

A preference for a status interpretation of the display does not stop us from modelling it using status change events, but does influence our perception of those events. A slightly more profound worry about intermittent display events is whether we should consider using a model that appears to sanction user commands with no feedback. Any event-in event-out model should include an *interactivity condition*. This would require that each user input event is followed by at least one system output event.

10.4.2 Result – status or event?

We now turn to the result component. The result has a much more immediate interpretation as an event. Certainly in the classic example we have used, the word processor, the result, i.e. the printed document, occurs at some (infrequent) point in the interaction. We can, of course, ignore here the time it takes to do the printing. It makes sense therefore to have the result as an event that occurs when the appropriate print subdialogue is complete. Similar arguments would apply to a text editor which works on a copy–rewrite basis. The alterations the user has performed are written to the file system only at the end of the interaction, or at specific periods when the user asks for the file to be written.

On the other hand, there is frequently an idea of the potential final result. So, when we use the word processor, we know that there is some document that *would* be printed if we issued the appropriate commands. This has a validity at two levels. On the one hand, there is the internal state of the system, which will certainly have some component corresponding to this potential result. On the other hand, from the users' point of view, at any point in time they will have a fairly strong idea of what the potential result will be. Thus both from the system's and the user's perspective the result is status-like.

We can link these two concepts of the result if we have some notion of a "normal" mechanism for obtaining it. For instance, a word processor may have a PRINT command, or a text editor a SAVE. We would regard the potential result of these systems to be the physical result when the appropriate command was issued. Breakdowns can occur in several ways. There may be more than one "normal" mechanism for obtaining a result. For instance, the text editor may have an EXIT which saves the file, but also a QUIT command with no save. In such cases we have to consider that one method of obtaining a physical result is more "normal" than the rest. Furthermore, the form of the final result may be determined intrinsically by the exact dialogue by which it is obtained. The print subdialogue of many word processors includes the choice of many important layout parameters such as page length, or multi-columning. In these cases, any notion of the potential result as status can be only an abstraction of the full event result.

These last two points are to do with an ambiguity over the notion of the potential result. If the user and the designer differ in their understanding of this notion, breakdowns in use are likely to occur. Within the limits of the user's perception, we can assume that the user and system agree about the current state of the display. There is no such guarantee for the potential result. We rely on the display to give appropriate cues for them to synchronise. We recall that a large part of Chapter 3 concentrated on this issue of what we can tell of the result from the display. Of course, users do not notice everything on the screen, so that even if the display has sufficient information on it to perform this synchronisation it may not be salient. In empirical studies of a reference database system at York, it was found that a significant class of error could be attributed to a disagreement between the user's and system's idea of the current result. (ce)

So where have we got to? Physically, results tend to be events but conceptually they are often best described as status. An important issue for any design is ensuring that the two views are in agreement.

10.5 Status inputs

In this section, we consider different models of status inputs. These models are related to and express different complexities of interactive behaviour of status inputs. As the most common status device is a mouse or other pointer device we concentrate on these, but much of the discussion is valid for any status input device.

We have already noted that we could include status inputs in the τ-PIE framework simply by adjoining a status to each time period. Let us now expand on this. At any instant we will assume that there is a current value for the status device and at most one event input. The event inputs we will take from a set of commands C and add a tick τ for the instants when there is no event. This is exactly as the τ-PIE. We then add the status inputs from a set Pos (for position). Thus the user input at each instant is a pair:

$$C_\tau \times Pos$$

where

$$C_\tau = C \cup \{ \tau \}$$

The user's input history is then a sequence of these:

$$H_{Pos} = (C_\tau \times Pos)^*$$

Just as in the τ-PIE we can define the behaviour either using an interpretation function over histories, or as a state transition function *doit*:

$$I_{\mathbf{st}} : H_{Pos} \rightarrow E_{\mathbf{st}}$$

$$doit_{\mathbf{st}} : (C_\tau \times Pos) \times E_{\mathbf{st}} \rightarrow E_{\mathbf{st}}$$

We obtain the display and result component from the minimum state $E_{\mathbf{st}}$ in the normal way. By interpreting some abstractions of $E_{\mathbf{st}}$ as event outputs and others as status we could obtain the complete possibilities of event/status input/output; however, for the purposes of this section we are interested only in the input side, and will ignore any distinctions in output.

An additional distinction we need to make on the input side is between those event commands which are physically connected to the status device and the rest. We will call these sets of commands *Button* and *Key* respectively, because when we consider the typical mouse and keyboard setup the mouse button events are clearly more closely connected to the mouse position than the keyboard. Such a distinction has a degree of subjectivity about it, but is not too difficult to make. There are awkward cases, as for example when a shift key on the keyboard is used in combination with a mouse button click. I personally find such combined keyboard–mouse chords rather abhorrent, but they are certainly common. If a system *must* have such events, they would belong in the *Button* class. The important thing is that this is a *physical* connection between the commands and the status device; there may or may not be a corresponding *logical* connection. However, it is precisely the mismatch between physical and logical proximity that I dislike so much about keyboard–mouse chords and which will be discussed later.

This model is now capable of expressing virtually any input behaviour. The properties which we describe in the rest of this section can be framed over this model. Sometimes, rather than doing this directly, we can develop a more specific model for a class of behaviours which we can relate back to this general model of status input. We start with those systems, or parts of systems, which are least dependent on status information, moving on to more and more complex status dependence.

10.5.1 Position independence

The simplest thing to do with status information is to ignore it! If the status is ignored throughout the whole system, we can just use the PIE or τ-PIE model. This is the situation for applications boasting "no mouse support", but is not very interesting.

Typically, some parts of an interface are position independent, in particular the keyboard inputs. We can make position independence a property of commands, by simply demanding that the interpretation of the command does not depend on

the current value of the status input. In the case of mouse-based systems a command is position independent if it has the same effect no matter where the mouse is.

We can express this property using the model described above:

position independence:
$\forall p, p' \in Pos, e \in E_{\mathbf{st}}: \quad doit_{\mathbf{st}}(\, c, p, e \,) \quad = \quad doit_{\mathbf{st}}(\, c, p', e \,)$

Within a single application, the same functionality may be obtained using position-independent or position-dependent commands. For instance, in Microsoft Word4, the PRINT option can be selected by a position-independent keyboard accelerator or by a position-dependent sequence of mouse clicks over menu options. The latter are position dependent because the interpretation of the mouse clicks is determined by the menu selection over which the mouse pointer lies.

As we have noted, if all the commands were position independent the system would be rather boring (with a few caveats) from a mouse's perspective. However, for some commands, and in particular the keyboard commands, position independence seems positively beneficial. Position-dependent commands may obviously require quite precise hand–eye coordination. This coordination of the mouse position with input which is physically remote from it seems rather questionable. One requirement we may therefore want to make of status inputs, is that all commands in *Key* are position independent.

Some windowed systems apply a "click to type" paradigm. In order to select a window for keyboard input some definite action must be taken, either clicking a mouse button over a specific part of the window's border, or just anywhere on the window. Other systems use instead a "focus under the mouse" paradigm, whereby the window which is under the current mouse pointer at any moment is the active one to which all events, including keyboard events, are addressed. In such systems, every event is effectively position dependent. If we look at the applications themselves, they frequently apply position independence for the keyboard, but this property is "spoilt" by the environment. The danger of such systems is that while the user's attention is on the keyboard, an inadvertent nudge against a pile of papers may unintentionally move the mouse and redirect the input. On the other hand, the movements required are often gross and it appears a relatively benign form of position dependence. We will see in a while that by reinterpreting the position status, the model can be adjusted to leave the keyboard commands themselves position independent. This adjustment will still preserve an element of warning about the behaviour, but will centre this on the mouse movement rather than on the keyboard. This seems to correspond to one's intuition about the situation, that it is not the application or the keyboard's "fault".

10.5.2 Trajectory independence

Now we shift our attention to events that are dependent on the position. An important subclass of these comprises events which depend only on the *most recent* position.

Let us consider a mouse-based word processor. At the top of the screen it has a line of buttons. Clicking the mouse over these uncovers further menus. As we move the mouse pointer over menu selections they are highlighted and we make a choice using a second click. Text is highlighted by depressing a mouse button over one end of the required portion, dragging the mouse to the other end, and then releasing the button. The highlighted text is operated on by various of the menu and keyboard commands. The text entry itself is position independent.

Notice two things from this description:

- As the mouse moves over the menu selections and as the mouse drags out the text selection, the display varies continuously with the mouse movement.
- When a menu selection is made, or when the text is selected, it is only the position of the mouse when the event takes place that is important. The intermediate mouse positions do not affect the final state.

The second of these says that the final state of the system can be determined by looking at snapshots of the position at the moments when events occur. If we define the *trajectory* of the mouse, as the history of exact movements of the mouse between events, then the second condition says that the commands in the word processor are *trajectory independent*.

Although the trajectory is not important in determining the final state, the first of the two observations reminds us that it *is* important for the user. It is the detailed feedback from the intermediate states which makes the mouse both usable and enjoyable. Indeed, constant feedback is the very heart of interactivity.

A simple statement of trajectory independence might look something like this:

$$\forall p \in Pos, e \in E_{\mathbf{st}} : \quad doit_{\mathbf{st}}(\ \tau,\ p,\ e\ ) \quad = \quad e$$

This will not do, however, since the display is determined from $E_{\mathbf{st}}$. We need to distinguish the part of the state that is purely to do with feedback from the functional part. We have already made similar distinctions in earlier chapters:

$$E_{\mathbf{st}} \quad = \quad E_{feedback} \quad \times \quad E_{functionality}$$

The trajectory independence condition would then be applied to $E_{functionality}$ only. We could, of course, "cheat" and choose the $E_{feedback}$ to be *everything* and the $E_{functionality}$ to be empty. This would make the system totally, and vacuously, trajectory independent. We can protect ourselves from such inadvertent cheating

by demanding that $E_{functionality}$ has at least sufficient information to determine the result mapping.

This split is similar to the layered design we described in Chapter 7. Here though, the functionality we wish to capture in the inner state is more than in E_{obj}. That was supposed to be the state pertaining to the actual objects of interest, ignoring their representation. Here we would wish to include features such as the position of the menu on the screen in order to interpret the mouse position when a button was pressed. In other words, this is really a further layer outside the models we were considering there.

Rather than looking at predicates over the general model, we can look at a simpler model which implicitly captures trajectory independence. This will also bring us back to something which can be directly related to the models used in Chapter 7. This model will apply only to systems which are *totally* trajectory independent.

10.5.3 A model for trajectory independence

The crucial property of trajectory independence is that the long-term behaviour of the system is determined only by the status at the moment when events occur. So we first look only at those instants. We have an internal state E_{ti} that is updated at each event. The update function makes use of the event (from C) and the status (from Pos):

$$doit_{ti} : \quad C \times Pos \times E_{ti} \rightarrow E_{ti}$$

The result is a obtained as a mapping from E_{ti}:

$$result_{ti} : \quad E_{ti} \rightarrow R$$

This differs from the general status input model in that we have dropped the τ events which signified non-events. This is because we are looking only at the instants when events occur. In fact, so far it is exactly like a red-PIE, with command set $C_{ti} = C \times Pos$. However, the display component will differ somewhat from that of the red-PIE.

The first of the two observations we made earlier was that the value of the status input was constantly reflected in the display. In order to capture this we make the display a function not only of the current state (from E_{ti}) but also of the current position (from Pos):

$$display_{ti} : \quad E_{ti} \times Pos \rightarrow D$$

So, in the case of the word processor above, in the state when the pull-down menu had appeared, the display would be of the menu (dependent on state) with the entry under the mouse highlighted (dependent on status).

By separating out the event updates from the status feedback, we implicitly capture the position independence of the model. It can be related back to the full status input model by setting:

$$E_{\mathbf{st}} = E_{ti} \times Pos$$

$$doit_{\mathbf{st}}(\tau, p, (e, p')) = (e, p)$$
$$doit_{\mathbf{st}}(c, p, (e, p')) = (doit_{ti}(c, p, e), p)$$

$$display_{\mathbf{st}} = display_{ti}$$

$$result_{\mathbf{st}}(e, p) = result_{ti}(e)$$

This simply adds the current position as part of the state. However, both the result and update parts of the model ignore this part of the state. In other words, it is just a simple way of achieving the split described in the previous subsection. This relationship between the trajectory-independent model and the full model is rather like the embedding of steady-state behaviour in the τ-PIE model.

We have already begun to tie this model in to the red-PIE. We can deal with the display component in two ways. One is to set the display from the PIE point of view equal to the *function* from *Pos* to *D*:

$$D_{PIE} = (Pos \rightarrow D)$$

$$display_{PIE}(e) = \lambda p \; display_{ti}(e, p)$$

This is not a totally indefensible position, but talking of the display as a function may appear a trifle odd. Perhaps a more natural approach is to consider an abstraction of the display. For example, in the word processor, we can think of the display with the contents of the menu items displayed, but ignoring the highlighting. In order not to lose important information, such as the fact that *something* is highlighted, we may need to add a little extra to this abstract display, but on the whole this would probably correspond to how a user might describe the display after an event. Again this reminds us of the way we can abstract away "awkward" real-time parts of the display (such as a clock) in order to obtain a system with steady-state behaviour.

So, we have not only produced a simplified model which describes trajectory-independent systems, but we also have the means to fit this class of status input systems into the layered models of design introduced earlier in the book.

10.5.4 Spatial granularity and regions

We have just considered *when* positional information might be used. We will now move on to *what* part of that information is used. To begin with, let us think about a simple CAD system. Like the word processor it has pull-down menus, and perhaps a fixed menu of shapes it can draw as well. Instead of a text window

it has a graphical area with points, lines, circles, etc. These shapes are drawn by selecting the appropriate tool icon and then clicking at the appropriate anchor points in turn: so to draw a line we would click first at one end and then the other. We could allow movement and reshaping of existing shapes by depressing a mouse button over appropriate anchor points and then dragging before releasing the button.

This could easily be cast into the trajectory-independent framework above. What we need to concentrate on here, though, are the differences between the icon selection and the point selection:

- *Size* – There is an obvious difference in size. The icon is just a bigger target and therefore requires less fine hand–eye coordination. The problems with achieving such fine control are often addressed by including grids for initial plotting and by having zones of attraction around the anchor points for later selection.
- *Semantics* – The use of positional input to choose the icon is just a way of indicating which icon is of interest. The same object could have been achieved with a keyboard accelerator or by speaking the name of the shape. With the anchor points, however, their very nature is positional, so the semantics of the anchor points have a close correspondence to the form of input.

These two differences are somewhat interlinked. Intrinsically positional parts of an interface tend to be fine grained, although there are grey areas. Text selection is just such an area, which is why we switched to CAD as an example! If we consider only granularity, text selection would lie somewhere between icon selection and point selection, and we might just regard characters as small icons. However, the location of characters within a document is intrinsically positional, and hence we should class it with point selection when we consider semantics.

There are obvious reasons for using the mouse for intrinsically positional features, but why is it used for less semantically necessary ones? In a CAD system where the mouse is used extensively for object-level manipulation, it makes ergonomic sense not to shift the user between input devices. This does not apply to a word processor, where typing is a major activity. In fact, in many mouse-based word processors keyboard accelerators are available in order to short-cut the use of the mouse and menus. In such systems the major reason given for the use of menus and icons is cognitive: it is easier to recognise than to remember, thus clicking over an icon that says PRINT (and perhaps has a suitable picture on it) is easier for the user than remembering whether to type PRINT or LIST. A further related reason is to increase the effective input bandwidth. We recall the mismatch in bandwidth between input and output. Iconic interfaces and menu-based systems are one way of using the output bandwidth effectively to increase the input bandwidth. This is especially obvious in a file-system browser, where clicking on a file is an apparently more effortless

task than typing the appropriate filename. This stretching of bandwidth is connected to the use of context in determining possible user inputs. The selection of a particular file icon, for example, is made easier because the directories or folders of interest are displayed. In terms of information flow, this is a form of clever coding which makes use of the redundancy of the user's input language. Non-graphical systems use similar contextual devices such as the idea of a current directory, current drive or current active file, as well as the use of "wildcards" in filename specification. Graphical systems extend this by allowing multiple simultaneous contexts, and by making the "coding" more easily discernible.

Whatever the reasons for the use of position dependence, if the granularity of the target is quite large then the hand–eye coordination required is far easier. For parts of the system, such as icons and menus, where the mouse can be anywhere in an area of the screen, we can factor the dependence of the system on the position by defining a *region* mapping:

$$region: \quad Pos \rightarrow Reg$$

For a particular programming environment, the elements of region might include menu buttons "File", "Edit", "Options" but also all-embracing things like "Program window". Having defined such a mapping we can distinguish those commands which are merely *region dependent*, like menu selection, from those which require fine grain positioning, like anchor point plotting:

region dependence:
$$\forall p, p' \in Pos: \quad region(p) = region(p') \Rightarrow doit_{ti}(c, p, e) = doit_{ti}(c, p', e)$$

A comparison of the numbers of region-dependent and fine-grained commands could form a good measure of the motor control required for an application. One could not use such a measure blindly, however, as one should look at the semantics of the commands and whether fine-grain control is appropriate because the feature is intrinsically positional. Again, one can cheat. One could say that every pixel in a graphics window was an individual region. This would fail the semantics test, as the pixels, although discrete, represent an intrinsically positional world. For the same reason, although it might be appropriate to factor text positions (possibly using *back* maps from the pointer space projection), they are an intrinsically positional attribute and should not be regarded as lots of screen regions.

In the examples given above, the intrinsically positional elements have all been at the application object level whereas the region elements have been interface objects. This is not always the case. Application objects frequently contain discrete subobjects which may be referred to by screen region. Also there are positional interface objects such as a scroll bar.

10.5.5 Trajectory dependence

We now move on to the final category of status input behaviour, *trajectory dependence*. This is where the outcome of an interaction depends on the precise path of the status device between events. The most obvious example of trajectory dependence is in freehand drawing or spray cans in graphics packages. In some ways this is more than an example and is archetypical of trajectory dependence in the same way that point plotting is archetypical of fine-grain position dependence. Freehand drawing is trajectory dependent not because a designer thought that it was a good idea, but because its semantics demand a trajectory approach. Non-trajectory-dependent forms of freehand drawing would be obscure to say the least.

It is frequently the case that drawing or painting is activated only whilst a mouse button is depressed. Thus the period for which fine hand–eye coordination is required is marked for the user by the conscious act of holding down a button. Furthermore, the button is on the device which requires the control: that is, the physical and logical proximity coincide. This "pen-down" paradigm is often used elsewhere, both in other trajectory-dependent situations, such as gesture based input, and also in trajectory-independent contexts such as window movement. To describe such situations, it is easiest to regard the holding down of the mouse button as another status input with a simple on/off value. We can then ask for trajectory independence when the button is up.

Freehand drawing and the like are not the only examples of trajectory dependence. We have already mentioned gesture input but there is a further class of trajectory-dependent situations based around revealing screen objects. A plethora of menus, scroll bars, icons, etc. may soon clutter up a screen, leaving little room for "real" information. Many designers choose to conserve screen space by hiding various features which appear only after some user intervention. Sometimes this intervention is of a trajectory-independent nature, perhaps clicking on a menu bar which makes a pull-down menu appear. However, sometimes it is trajectory dependent. For example, some window managers uncover any window over which the mouse moves.

Line drawing intrinsically requires trajectory dependence. These other features do not. The intention is presumably to make the user's job "easier" by not requiring an explicit mouse event. This is a very dubious practice and requires careful consideration. It is a problem firstly because of the extreme level of hand–eye coordination required. Position-dependent commands require fine control for only the fraction of a second that it takes to depress or click a button. Trajectory dependence requires consistent control over some period. Secondly, the problem is further exacerbated by the lack of *proportionality* which is typical of trajectory dependence.

There are several forms of proportionality, and here I am thinking of the ability to undo erroneous inputs. We have dealt with various types of undo properties, and the ability to undo easily is regarded as an important feature of direct manipulation systems. (Shneiderman 1982) Not only should users' actions be undoable, but as a general rule there should be some proportionality between the complexity of an action and the complexity required to undo that action. Any true measure of complexity requires psychological insight, but a rather crude interpretation is all that is needed here.

Consider a trajectory-independent system. Imagine the user is moving the mouse and no other events occur. The user wishes to move to a position p but gets to p' instead. All the user needs to do to correct the situation is to move the mouse back to p. Because the intermediate positions are not important for the long-term development of the system, the mistakes in positioning have no long-term effects. Note that mouse movement is sufficient to undo mouse movement, and that small positional errors (from p to $p + \delta p$) can be recovered using small movements (δp).

10.5.6 Some trajectory-dependent systems

With a little understanding of the problems of trajectory dependence, we can go on to look at a few examples. Several windowed systems have "walking menus". Some options in a menu have submenus associated with them. Instead of having to select the relevant menu item to make the submenu appear, the user has simply to slide the mouse off the side of the item. The system is position dependent, as the same screen position may be associated with several submenus depending on which main menu item the user moved off. However, if the user slides into a submenu unintentionally then the mouse can be slid back again to regain the main menu. Thus, even though the system is trajectory dependent it still retains the proportionality properties that mouse movement can undo movement errors, and that small errors have small corrections. Further, the corrections are the opposites of the movements that caused the problem, so there is a further naturalness about the situation.

The Xerox InterLisp windows (cf) make use of appearing scroll bars. This is a deliberate policy of reducing screen space usage. With menus there is little alternative. It would not be possible (let alone desirable) to have all the menu options permanently visible and the issue is purely the appropriate method of revealing them. However, it is perfectly feasible, and in fact quite common, to have scroll bars permanently associated with each text window. The mechanism used by InterLisp windows is as follows. If the mouse moves off the left-hand edge of a text window a scroll bar appears under the mouse, beside the left edge of the window. If the user continues to move to the left and moves off the edge of the scroll bar (or in fact, anywhere off the scroll bar), it disappears again. How does this measure up to the proportionality tests?

There are several classes of problems so let us consider just a few. One mistake a user might make is to slip off the left-hand edge of a window by accident. The scroll bar appears, but a movement right again back into the window corrects this. Again, the recovery is with the mouse, proportional to the error and in a natural inverse direction. If, on the other hand, the mouse was in the scroll bar and slipped out to the left, the scroll bar would disappear. In order to recover the user would have to move a longer distance to the right to get back over the window, then move left to recover the scroll bar. The recovery is still purely by mouse movement, but the complexity of the correction has grown somewhat, and the movements are less natural. A similar scenario occurs if the mouse is accidentally moved to the right over the left border of a window. Trying to correct by moving the mouse back the way it came would then make the scroll bar appear, incidentally covering up what was probably the focus of attention. Again recovery could eventually be made using mouse movement alone, but the correction is complex and an attempt at a natural recovery leads to further complications.

In the case of InterLisp, the complications arising from trajectory dependence obviously require extra care and fine motor control to avoid mistakes. However, the property of movement as recovery for movement is preserved, and perhaps the sort of mistakes noted are rare. The advantage is that by making the scroll bars appear, they can afford to be larger and thus require less fine hand–eye coordination. There appears to be a trade-off between the two. The appropriate design choice is not obvious from formal arguments alone. What we have done is focus on possible problem areas.

Our final example of trajectory dependence comes from a version of the Gem environment. (cg) The system displays a menu bar along the top of the screen. Instead of requiring the user to depress a button over the desired option to obtain the pull-down menu, the system "saves" the user the bother of pressing the button by making the pull-down menu appear as soon as the mouse enters the region of the menu bar. The user can then move the mouse over the desired selection and click on it to make their choice. This is somewhat similar to the previous examples, and might be thought to be no more and no less a problem. If we begin to consider errors and their correction, however, it is seen to have distinct problems.

When the pull-down menus appear, they do not disappear until either one of the menu items has been selected or the mouse is clicked elsewhere. So, if the mouse is accidentally moved over a menu bar title, the mistake cannot be corrected without an event input. Further, the mouse must often be moved quite a way to get it somewhere where extraneous mouse clicks do no harm. That is, it disobeys virtually all the proportionality properties we have mentioned. Is this just a rather extreme example which rarely occurs in practice? I have certainly observed this as a problem myself, and have also watched both children and

novice users becoming stuck with a pull-down menu they do not know how to get rid of.

So what did the designers gain by this decision? Apparently they saved the user from pressing a mouse button, and perhaps made their interface slightly more distinctive from similar products. It is fairly obvious as soon as one considers trajectory dependence that this is a potential problem area. By taking this into account, this confusing design error could have been avoided.

10.5.7 Region change events

Just as position dependence can be simplified to region dependence, a similar simplification may occur with trajectory-dependent commands. Both of the examples from the InterLisp and Gem interfaces were dependent only on the regions through which the mouse moved. In the InterLisp case these regions were windows and scroll bars, and in the Gem case they were the menu bar and pull-down menu options. We could capture this form of region–trajectory dependence using the full model something like this:

$\forall h, h' \in H_{Pos}$ **if** h and h' are "region equivalent"
then $I_{\mathbf{st}}(h) = I_{\mathbf{st}}(h')$

where two histories are *region equivalent* if they are the same length and agree exactly at the events and have the same regions at other times:

h region equivalent to h' **if**:
(i) $h = h' = null$
(ii) $h = (c, p) : k$
$h' = (c, p) : k'$
and k and k' are region equivalent
(iii) $h = (\tau, p) : k$
$h' = (\tau, p') : k'$
$region(p) = region(p')$
and k and k' are region equivalent

In both examples, the regions change during the interaction, complicating matters further; the *region* mapping must take into account the current state of the system. We could go back and add this into the above definition. There is, however, a better way to go about it.

We recall that changes in status are often interpreted as events. The clock hands pass 6 o'clock and I turn on the radio for the news. A continuous, status output of the clock becomes an event for me. In a similar fashion, the continuous movement of a mouse, or the change of any status input, can give rise to system events. In the InterLisp example, as the mouse moves over the window

boundary, we can interpret that as an "uncover scroll-bar" event. Similarly, we can regard the movement of the mouse over the Gem menu-bar as a "pull menu" event. The events occur when the mouse moves between regions, and instead of looking at region–trajectory dependence with its obscure definition, we can talk about *region change* events.

These events lie somewhere between the status input device itself and fully fledged user event commands. By regarding these changes in status as events we can discuss the behaviour of systems such as InterLisp and Gem in an "eventy" rather than a status fashion. We do want to retain the distinction between these events and more explicit user events such as keypresses and mouse button clicks. The proportionality properties discussed above become *reachability* requirements on the subdialogues involving region change events, and in general we may want to impose more stringent properties on this class of event. In addition, the regions typically depend on the current state, so the region change events will necessarily have a language, whereas the explicit events are more likely to be unconstrained.

With status change events in mind, we can look again at those windowing systems which direct keyboard input at whatever window the mouse is in. We noted at the time that this makes all commands, both from the mouse buttons and the keyboard, *region dependent*. This is unfortunate as it means that although the keyboard commands may be completely position independent for each application individually, this independence is lost when looking at the system as a whole. If we now regard the movement of the mouse over window boundaries as a "select this window" event, the state of the system then changes in response to the event and the keyboard becomes again position independent. This outlook agrees with our intuition about the system: any oddness is not the "fault" of the keyboard commands, but is because of moving the mouse. Our focus is shifted from the keys towards the region changes, where it belongs.

So, if our focus is on region change events, what questions should we ask about them? One obvious thing to ask is, did the user mean it? The problem with directing keyboard input "at the mouse" is that attention may not be on the mouse, and any movement, and consequent region change event, may be unintentional. Of course, we cannot guard totally against accidents: it is as easy to hit the wrong key as to bump the mouse. However, we can look at a status change event in its expected context of use and ask where the user's attention is likely to be, and thus how often to expect accidents.

Another way to look at this type of error is as a mismatch between the perception of events by the user and the system. If I hit a key, the action of striking it forms an event for me and the reception of that character is an event for the system. On the other hand, if I happen to move my mouse over the pixel at coordinate (237,513) which is totally unmarked, I will not regard this as an

event; if the system does so, then a breakdown will occur. The situation is worse if I merely joggle the mouse with my elbow!

In summary, region change events are helpful in describing some types of behaviour. This is recognised at an implementation level in that many window managers will generate events for the programmer when the mouse moves between certain types of screen region. It has also become clear that region change events are a source of potential user problems and should be scrutinised closely during design. In particular, when region change events are contemplated we should ask ourselves whether the events that are recognised by the system agree with the events which are salient for the user.

10.5.8 Complexity and control

We have discussed several classes of model for dealing with different types of positional input system. The complexities of the models correspond to various degrees of hand–eye coordination. The various types of event are summarised below (*fig*. 10.1), set against the degree of control required of the user.

Event class	Motor control required
Position independent	Ability to strike correct keys, possibly touch typing
Region dependent	Ability to retain mouse within target
Position dependent	Fine positioning for duration of "click"
Region change events	Ability to move mouse over target boundaries
Button-down trajectory dependence	Fine positioning for controlled periods
Full trajectory dependence	Continuous fine control
Time–position dependence	....

figure 10.1 *control required for event classes*

An additional category has been added of *time–position dependence*. This is to cover those systems that depend not only on *where* the mouse has moved, but on *how long* it has stayed there. There are probably few examples of this with mouse-based systems outside the games world, but it is more common in other systems. The accelerator pedal of a car has this property: where you get to and how fast depends on how far and for how long the pedal is held down. The same

is true for the steering wheel and many other controls.

On the other hand, more complex forms of positional inputs can be used to obtain special and useful effects. The formal distinctions between these classes of systems can be used to warn about possible complexities in the interface, but it is a matter of judgement and perhaps empirical testing how to make the trade-off.

Also, although fine control can be a problem in an interface it can also be an enjoyable feature. Perhaps many people find mouse-based systems more enjoyable to use precisely because of the demands on their skills. Whereas event-based systems have a clerical nature, status input gives more a sense of hand-craft.

10.6 Communication and messages

Virtually all our discussion has been about single-user systems. The models of multiple windows drew on an analogy with multiple users, and this led on to recognising interference between users as one of the sources of non-determinism discussed in Chapter 6; however, this was a minor thrust in both chapters. The emphasis, such as it was, dealt with how to avoid such interference between users. This approach is appropriate in a multi-user system with shared resources but in which each user works independently. For shared or cooperative work a different perspective is required.

There is, of course, a lot one could say about cooperative computing systems and it is one of the current growth areas in HCI research. For the rest of this chapter we deal briefly with some specific aspects which relate to the concepts of event and status.

To begin with, let us think about two contrasting communication mechanisms. First, there is a traditional email system. Each user has a "mailbox". Other users can send messages which should eventually, via various networks, find their way into the recipient's mailbox. When the users look at their own mailbox they find any messages that have been sent to them. Additional facilities offered typically include distribution groups and aliases (as the addressing systems are far from clear) to help send messages, and various forms of visible or audible indicators when mail arrives.

The second type of system to consider is where the model is of a shared information base. At the simplest, users of a shared file system can use this as a communication mechanism. Perhaps two researchers working on a shared paper will read each other's contributions simply by looking at the appropriate files. Other systems have been designed specifically to encourage information exchange and cooperation. Some are based on hypertext techniques, such as for instance KMS (Yoder *et al*. 1989). The whiteboard (blackboard, chalkboard)

metaphor has also been used (Donahue and Widom 1986, Stefik *et al*. 1987), and also simple screen sharing. (ch)

These two extremes represent two styles of communication:

- Messaging
- Shared information

The astute reader will have already guessed how these relate to the theme of this chapter, but before discussing this we will examine a few other features of this distinction.

10.6.1 System and user cross-implementation

What a system provides and what users do with those facilities are, of course, very different. Most communication systems are built on top of an underlying messaging protocol between the users' computers. Even where the users share a single file system, their workstations will typically communicate with the file server by network messages. This level of implementation is usually hidden from the users. The network messages may be used to implement an apparently shared data space, and even where the user level model is of messaging this will be built on top of the underlying network in a non-trivial manner. In a similar manner, users are proficient at producing social mechanisms for "implementing" one protocol upon another.

Think about two chess players swapping moves by post. The postal system is a messaging system, but is used to keep their respective boards consistent. That is, they maintain a shared information space. The transfer of files by email is a similar situation. The users want to share the information contained in their respective filing systems and use the email messages to implement this. It is interesting that even where users share a common file system they may still use email for this purpose. This is presumably because of the difficulty of giving the appropriate permissions or specifying the file names.

Social implementation techniques work the other way too. Users of shared information systems make use of drop areas: they set aside parts of the information space for each user and simply add their messages to this area. (ci) This mechanism is a direct parallel to the way that many systems implement internal email. Also, bulletin boards may contain a permanent record of all transactions but many users will read only the most recent entries and never consult old entries. (cj) They thus become simple broadcast message systems. Again, personal columns in newspapers are messages between individuals which make use of a widely shared information base (the newspaper).

Even though users are quite capable of shifting the communication paradigm supplied by their system, these examples do suggest that support for the actual tasks performed may be beneficial. On the one hand, we may ask whether drop

areas in a shared information system form an adequate message system. On the other, it is clear that if we want to transfer information, such as files or more complicated structures, sending messages is not the way to do it.

10.6.2 Attributes of messages

If we want to know how to design communication systems that meet users' needs, we should consider some of the different attributes of messages and shared data. Not all systems need be as polarised as the examples we have given, and the way users implement different social protocols can make it hard to distinguish the crucial attributes that make a system suitable for a task or not. Attempting to uncover the central attributes that differentiate these mechanisms can help us understand this suitability.

If we contrast direct speech with, say, a book or filing cabinet, we see an obvious difference in *persistence*. The spoken message is ephemeral: its permanent effect is in the way it has affected the users' actions and their memory. The shared information in the book and the filing cabinet is much more long lived. This difference in persistence between messages and shared data seems to be fairly characteristic. Email messages tend to be read once only, as compared to a shared database that is intended to be a perpetual resource. However, the distinction is not as clear-cut as all that: both electronic and paper messages are frequently filed permanently, and spoken meetings are made permanent by the use of minutes or notes. In fact, it is a general feature of formal bodies that ephemeral messages become permanent corporate information.

There are two major reasons for recording messages. The first is that they may contain information that is required on a long-term basis. Arguably, if that is the case, then that part of the message was partly a sharing of information that is implemented by the message. A second reason is as a record of a conversation, either to re-establish context when new messages are received, or to serve as a legal or official record in case of dispute. In the former case only the last few messages are required, so the messages are still ephemeral. In the latter case, we have a meta-communication goal, and it is not surprising that it has strange characteristics; the medium through which we communicate has become the object of higher-level tasks.

Other attributes that can help distinguish messages from shared data are *ownership* and *identity*. Shared data tend to remain the property of the originator or may have some sort of common ownership. The *same* data object is available to all. Messages tend to be transferred between parties, and the recipient either gets a copy of the original item, or the sender loses their copy. The typical example of this is the letter. The sender can retain a copy or send a copy, but the letter itself changes hands. Similarly, email messages become the property of the

recipient. Again there are exceptions, for instance, if I make lots of photocopies of this chapter to pass to my colleagues to review, but this is a typical example of implementing shared information using messaging.

We can lay out these attributes in a matrix, placing messaging and shared data in it:

		persistence	
		ephemeral	permanent
ownership	transferred	message	(a)
	shared	(b)	shared data

There are two gaps in the matrix, but are there any situations that fit in these gaps? Let us look first at gap (a), that is, transfer of long-lived information. Obvious examples of this in the real world are legal transactions, where the transfer of some written document carries with it the ownership of some asset. This could be seen to be a little stretched because of the link with physical objects. One example of a "pure" information transfer within this category is the passing on of a job. All the relevant information, both electronic and paper, is passed on to the new post-holder. Both these examples seem closer to a data-oriented view of communication. Any messaging that would be associated with the transactions would appear to be additional, even if sometimes within the same enclosure.

The other gap, at (b), is for ephemeral data in some form of shared or common ownership. This is a more difficult gap to fill. One could imagine information in a shared database with limited lifetime, for instance, a meeting announcement. However, the ephemeral nature of this information is related to real time and is very different from the ephemeral nature of messages. Messages are ephemeral because they are no longer needed once they have been read: that is, their timescales are subjective. The only examples of shared data with a subjective timescale that I can think of are items like electronic voting forms, or data items which are "ticked" off when they are read and are removed when ticked by everyone. The former have a quite complex dialogue associated with them, but are essentially message-like. The latter appear to be an example of shared data being used to implement broadcast messaging. A physical example of the latter would be the university lecture, but of course this would lack the guarantee of universal attention.

If anything, the permanence axis seems more characteristic than the ownership one, but all the off-diagonal options are rare or expose failings in the underlying transport mechanisms. Of course, a good shared information store would support many different types of ownership and ownership transfer. Further, it is easier to make permanent data ephemeral than *vice versa*, so the shared data

view seems to include messaging as a subcase. This is borne out to some extent by the experience that in shared data systems, the attempts at establishing messaging via drop areas seem more successful than the attempts to establish shared information using email.

So, is that all we need to know about messaging? Should we concentrate on producing effective shared information systems, as these can be configured to include all forms of communication? In fact, there is a vital element missing, and we need to return to the subject of this chapter, events and status.

10.6.3 Messages as events

As we noted earlier, the reader may well have noticed a similarity between the event/status distinction and messages and shared information. Messages happen, information is there. This is exactly the distinction between event and status. The message is ephemeral, because its effect is achieved once it has been received.

One day someone comes into my office and puts a paper to be read on a pile at the back of my desk. It effectively passes into my ownership and my personal workspace. However, let us elaborate on this in two alternative scenarios. In the first, I am away. The paper is left on the pile with a note on top inviting comments. Assuming I notice it there upon my return I will discard the note, but I may not notice it for a while. In the second scenario I am there: "here's a paper you may like to read" ... the paper is flung onto the pile.

In the first scenario, the "message" part of the transaction was ephemeral and transferred ownership, but it failed in its purpose. Because it did not achieve sufficient salience for me, the event of my receiving it was delayed, possibly indefinitely. The difference between the spoken message and the written note is immaterial: if the note had been placed silently on my desk because I was on the phone, it would still have achieved salience and would still have achieved its purpose as a message. To make matters really extreme we could contrast, on the one hand, the note being slipped silently onto my desk while I was in the room but with my back turned, with, on the other hand, it being put there just as silently but in full view. A snapshot of the room just before and just after would be identical, but in the latter case an event would have occurred and a message would have been passed.

10.6.4 Why send messages?

Why is the salience and event of reception so important? We must look at the reasons for the sending of messages.

The first reason is imperative. The knock on the door and cry "Fire!", and the Poll Tax demand are both intended to prompt action. The salience is necessary if we want the action to occur in time; however, the immediacy required in the two cases is different. Putting a letter through the door with the word "Fire!" on it would not be appropriate, neither would knocking at the door and shouting "Poll Tax".

Often, combined with this desire for action is a more subtle reason. If you know that your message was salient, you *know* that the recipient has read it. In addition, the recipient knows that you know, etc. That is, it achieves its effect by creating and transferring meta-knowledge. Examples of where this is important would include informing interested parties about a decision even when they have no opportunity of affecting it.

When combined, messages may achieve quite complex social purposes. For instance, not only does the Poll Tax demand prompt you to pay, but the fact that *you have received* the demand and have not paid makes you liable to prosecution. Letters sent to those who do not hold television licences which request confirmation of receipt have a similar legal entrapment function.

Speech-act theory (ck) treats a conversation rather like a game, with various moves open to the participants at different stages. Breakdowns can occur during face-to-face conversation due to misunderstandings, but remote conversations open up further possibilities. If I make a move in the conversation game and the other participant does not know about the move, then she may make a move which is perfectly valid from her view of the conversation but may be confusing or meaningless from my point of view. Thus the salience of my messages forms an important role in synchronising our conversation.

10.6.5 Appropriate events and user salience

Thus messages must give rise to events salient to the recipient. There are various ways by which users perceive events:

(i) An event-type system output, e.g. a beep or sudden flash of the screen.

(ii) An indicator on the display, e.g. a mail ready flag.

(iii) A change in the display, e.g. a mailbox size increase.

(iv) An indicator which is uncovered during subsequent interaction.

(v) A change in status which is uncovered during subsequent interaction.

Notice that (iii) and (v) are both cases where a change in status is interpreted as an event. There is, of course, the possibility that such status change events are *never* noticed by the user: perhaps by the time the user's attention is on the appropriate part of the display or database, they have forgotten what the previous state was. The eye is very good at noticing changes, so display changes will be noticed as long as they occur reasonably close to the user's focus of attention.

However, changes that are not immediately visible are very likely to be missed and so it would seem good policy always to have some persistent indicator rather than to rely solely on status change.

The same is true for the event outputs of class (i). These have the greatest immediate salience but, once missed, will never be noticed. Again, a permanent indicator will usually be required. These requirements for status records of events were discussed earlier in this chapter. A good example of an event indicator which combines both immediate salience with persistence is the Mac "Alert". These dialogue boxes disable the system entirely until they are responded to, hence they cannot be missed, yet they remain on the screen if the user is not present. They have other less fortunate properties, however. They interrupt other work, but the same could be said for a fire alarm. More seriously, they often demand that the user takes some action such as a decision before allowing the user to continue. Thus their purpose as an event, and a connected requirement for user intervention are conflated.

Status indicators of classes (ii) and (iv) only become events when the user notices them. Again, as we discussed early in this chapter, they may require some pattern of use to be sure of being noticed. For instance, my workstation has a flag on the mailbox which is raised when new mail has arrived (class ii). Periodically I look at the flag to see if there is any mail. If the flag were not there I would periodically have to use a mail command to examine the contents of the mailbox (class iv).

If the sender of a message wants to know that the message will be received then something must be known about the way the arrival of the message is signalled, and possibly about the habits of the recipient. If I know that someone looks at their mailbox only once a day when they log on to the system, I would not use email to ask them whether they can meet over lunch. When we discussed this issue earlier we noted that the granularity of events would be related to the pace of the task. Then we were thinking of automated tasks with a single user. A similar observation holds here though, that the pace of the conversation will influence the acceptable granularity of perceived events. This is why it is unacceptable to put a letter through the door warning of fire, whereas the task of paying taxes is measured in weeks (at least!), and therefore my infrequent checks on my doorstep are sufficient to make the letter an appropriate message.

We see then that in order to deal with a range of types of communication, we must not only be able to deal with information transfer and associated issues of ownership and persistence, but must also account for events. This is the key feature which mail systems possess but shared information lacks. Where users implement messaging using shared data they have to develop protocols for examining the appropriate parts of the data space. Adding features to a shared data model to raise appropriate events would be a major step towards producing a common framework for communication.

10.6.6 User status and system status

Finally, after considering the central role of events in communication, we return to status. The shared information entered by users as part of their communication is part of the system. A crucial part of face-to-face communication is the status of the other participants: whether they are attending, whether they want to speak, etc. Without such information it can be hard to establish effective communication. Once users are separated by a computer system they may not even know whether the other participants are at their terminals, let alone whether they are attending.

In Chapter 4 we considered techniques for capturing user task knowledge about the interdependence of windows. There is a similar issue here. Obviously, only information within the system can be presented to the user. However, the crucial status is about the users themselves: their attention, attitude, presence, etc. This is not immediately available to the system and cannot therefore be passed on to the other users. There are several possible ways of dealing with this problem.

First, we might ask the users to tell us what they are doing. This is likely to be an imposition and may be neglected. Still, if users know that the information they supply will be useful to other participants in a conversation, they may cooperate.

Alternatively, there may be information within the system which gives a reasonable indicator of the user's status. For instance, if the user has been typing recently then it is reasonable to assume that the user will be attentive to new messages, especially if they are accompanied by an audible or strong visual signal. Screen-savers work on precisely this principle, and thus the relevant mechanism may already exist on the system. A similar indicator is a screen lock, which implies that the user is away, but will return. These are both partial indicators of the user's status: the user may leave the screen locked (or turn the machine off) whilst still in the office. Similarly, the user may be attending to the screen but not actually typing.

It is possible to go further. Not only can existing system information be used, but also dialogues can be deliberately designed which expose aspects of the user's status. For example, in an experiment at York, each participant was required to select windows as part of their normal use of the system. These selections were designed to reveal information about the user's current activity. This information was then made available to the other participants.

10.7 Discussion

I have discussed status and events in many detailed circumstances, and it is time to recap on some of the properties that have been encountered. However, before I do that I would like to reflect on one last facet.

10.7.1 Events and dynamism

When we consider status it is (almost by definition) about *static* properties. It can be derived from a snapshot of the system at a particular moment. Furthermore, if we look at a time when there is no event, the status a little before and a little after is likely to be very similar. Hence status also emphasises continuity. Events, on the other hand, are at moments of change: they punctuate the dialogue, marking discontinuity and *dynamism*.

We can see intimations of this in previous chapters. The distinction between static and dynamic invariants has a similar feel. Static invariants relate various aspects of the status of a system. Dynamic invariants still talk about status, but limit the change in status when events occur. However, dynamic invariants do not tell us about what will happen at specific events, but provide only general restrictions on the sort of thing that can happen.

Both pointer spaces and complementary views told us something about the dynamism associated with particular events, but in rather different ways. For complementary views, the concentration on process-independent updates meant that we virtually ignored the *method* of change at an event, and instead looked only at *what* change occurred. This simplified matters considerably in what was still a tricky area; however, when we got to the end of the chapter and took a quick look at dynamic views a different perspective was required. We discovered that different sequences of updates could take the underlying data-base from the same start point to the same final value, and yet a different view was required. That is, the method of change became crucial. This led us to see the similarities with dynamic pointers and pointer spaces.

The pull function in a pointer space is a rich construct as it focuses on the *manner* of change. We went on to mention the concept of change information in general, of which the pull function and locality blocks were examples. Unlike the result and the display, which are status information, this information is associated with the event itself.

It is easy when thinking about concrete interfaces to concentrate on static aspects: at the surface level, screen mockups and perhaps at the semantic level, data structures. Dynamics are difficult to describe, both in written language and in the visual language of screen dumps. It is, however, the dynamics that are often crucial to the feel and effectiveness of a system. One of the points made in the pointer spaces chapter was that the dynamics of positional information are

typically not considered carefully enough in design. For instance, it would be easy to approach the design of a novel graphical hypertext with beautiful screen displays and subtle representations but not discover the problems of updating the links until it was implemented.

Events are obviously regarded as important in design and, in particular, dialogue descriptions are almost purely event oriented. It is, however, in linking the worlds of events and status that we are likely to come unstuck. The common approach is to look at the event as a state transition. That is, we reduce the event to its effect on status, rather than looking at it as important in its own right. Bornat and Thimbleby (1986), in their editor ded, went to great lengths to make the manner in which the screen is updated emphasise the direction of movement through the text. This is possible on character terminals which frequently have fast methods of scrolling upwards and downwards. Unfortunately, these insights have been all but lost in bit-mapped systems. Bit-map scrolling is often far too slow and the tendency is simply to repaint the screen. So, even at the raw interface, the event of movement is reduced to a change in status. This loss of dynamics is sad, and it is particularly unfortunate that an apparent improvement in technology should lead to a reduction in other aspects of interface quality.

10.7.2 Summary – properties of events and status

We shall now briefly review the various properties of status and events. The fundamental distinction is perhaps that events *happen* whereas status simply *is*. This was exactly the same distinction that we noted between messages and shared data, and the same issues tended to arise in both the single-user and multi-user cases.

Granularity and context

The way in which we regarded the ringing of the alarm at the beginning of this chapter depended on the timescale with which we looked at it. If we look at the ringing at a fine timescale it is a status, yet at a coarse timescale it is an event. The appropriate granularity depends on the task and the context: hence we must establish a contextual framework for statements about status and events. In a similar way, the context of communication determines what would be regarded as an effective message.

Status change

An important way in which events and status interact is that changes in status may be perceived as events. Just as events, when looked at closely, can often be interpreted as status, so can apparently continuously changing status, when examined, be seen as many change events. More important, however, even at the scale whereby the individual changes are not salient, certain changes of status

through critical values become, for the recipient, events. Examples of this include the hands of a clock passing a certain time, a mouse moving over a window and seeing a postcard appear on a noticeboard.

Salience

The "may" at the beginning of the previous paragraph was essential: changes in status are interpreted as events only when and if the recipient notices them. Similarly, objective events become subjective events for the user (or the system) only when they are noticed. Thus, in determining what are to be seen as salient events, we must have regard to the focus of attention of the recipient (and grab that attention if necessary). Further, for status change events it is often necessary that the recipient engages in some periodic procedure to scan the appropriate status. Sending an effective message requires not only that it should arrive, but also that the recipient will notice that it has arrived and thus act upon it.

Synchronisation

As events are partly subjective, there is the possibility that different partners in an interaction, whether a system and a user, or people in a conversation, will differ in their interpretation of an event. It is crucial that a message I send is not only an event for me, but that it becomes an event for you also. Our individual perception of the state of our conversation depends on mutual recognition of events; without this, we may diverge in our respective interpretations of where we are. This is especially important when silence is interpreted as a speech act, or where a message is intended to terminate a conversation. Synchronisation was also important when we discussed the way that the system may regard some changes in mouse position as events which the user considers inconsequential.

Dynamism

Finally, we have only just discussed the role of events in injecting a sense of dynamism. Talking via a pin-board is a far cry from a face-to-face conversation.

10.7.3 Conclusions

We have examined the role of status and events both in single-user interaction, and in the form of messages and shared data for multi-user cooperation. It is not surprising that similar features arise in the two areas, as we can, of course, regard the single-user case as that of a user in cooperation with the machine. In fact, it was a consideration of the parity between the two which began this chapter.

The most interesting thing about status and events is the interplay between the two. Unfortunately, it is hard to find models which adequately describe both. As we have noted, the models in this book largely fail in this respect, as do other formal models of interaction. This failing is not confined to HCI; similar

problems are apparent in general computing. For instance, models of concurrency tend to be totally event-based, ignoring status completely. On the practical side, database systems are almost entirely status-based and this is a major shortcoming when they are considered as vehicles for cooperative work. The only computing paradigm that could handle both to any extent would be *access-oriented* programming. (cm)

This lack of allowance for the range of status and event behaviours seems particularly unfortunate since our real-life interactions are typically mixed. An office worker will respond to memos and phone calls, consult paper files and databases, have conversations and smell lunch cooking in the canteen. A cyclist must be aware of road signs and road markings, traffic lights, the movements of other road users, and car horns. In addition, the cyclist will control the cycle's road position, use hand signals, and perhaps a bell. A factory controller will watch various status indicators, temperatures, pressures, and stock levels, whilst being ready for breakdowns, the arrival of bulk deliveries and knocking-off time.

In order to comprehend the complexities of the world, it is often necessary to simplify. However, this wealth of variety suggests that there is much to be gained from a richer understanding of event and status in interactive systems design.