Modelling status and event behaviour of interactive systems

Alan Dix	Gregory Abowd
School of Computing	College of Computing
Staffordshire University	Georgia Institute of Technology
Stafford, UK	Atlanta, USA

September 16, 1996

Abstract

Interactive systems involve both events which occur at specific moments (e.g. keystrokes, mouse-clicks and beeps) and more persistent status phenomena which can be observed at any time (e.g. the position of the mouse, the image on the screen). Most formalisms used for interactive systems concentrate on one aspect or another and may be asymmetric in their treatment of input and output. This paper classifies notations and models for interface specification by the way they treat status and event phenomena in their input and output. We use this to construct an model and associated notation which incorporates both. Specifying examples using this model highlights important design issues which would be missed if either status or event phenomena were not properly treated.

1 Introduction

Implementing and reasoning about any interactive system is a difficult and error-prone activity. Correct behaviour of a system from the user's perspective is vital as any error will destroy confidence in the system. Given the social fragility of groupware systems [21], even minor problems can be a disaster. To reason about the sorts of properties expected of any interactive system, we need a language that most naturally expresses the concepts of interest in that system. For example, in groupware systems, we need to describe multiple streams of control, multiple views of shared objects and also the distributed nature of the computation. In this chapter, we investigate the features of formal specification languages that promote their expressiveness for interactive behaviour. Specifically, we are concerned with the distinction between *event* and *status* phenomena at the user interface and how that distinction is useful for categorising specification approaches.

Status–Event Dichotomy

The dichotomy between event and status phenomena has been pointed out by the authors previously in a shorter version of this paper [3] and other works [14, 12]. Events are atomic, non-persistent occurrences in the world, that is, we sense that they happen at a particular point in time. Status refers to things that persist and we observe in the world, that is, they have a measurable value at any moment. A mouse click, or a beep indicating the arrival of a mail message are examples of events, while the position of the mouse cursor on the display or the position of the flag on the mailbox icon are examples of status information. There is a link between events and status; for example, the beep signalling the arrival of a mail message will often be associated to a change in the status of the position of the mailbox flag. As the mouse cursor moves across the boundary of a window, that window can be activated as the focus for further user input. As these examples show, events can trigger status changes and changes in status can trigger events.

In fact, the relationship is slightly more complicated. The idea of an event depends somewhat on viewpoint and timescale. If you consider the activities of a day, the ringing of your alarm in the morning is an event. If instead, you think of the act of getting up in the morning, the alarm is a status (ringing or not) and the critical events are when it starts to ring and when you press the button to stop it. The situation is similar in computer systems. Consider clicking a mouse button. At a physical level there are two events: down and up (or perhaps even more considering the de-bounce circuitry). From the user's point of view she simply clicks the mouse – a single event. Depending on the window manager and toolkit used, the program may see one 'mouse click' event or separate 'mouse-down' and 'mouse-up' events. This is an issue which we have explored in greater detail in earlier papers [13, 14]. However, the critical issue for this paper is that a user interface specification should *first* reflect the events and status as they are perceived by the user rather than those of the implementation of the system. The proper place to move towards the implementation perspective is during refinement, an issue we return to at the end of the paper.

Whereas this paper focuses principally on the formal aspects of the status–event distinction, perhaps the most significant feature of status–event analysis is that it forms a conceptual and analytic bridge between formal and informal understanding of problems. In particular, status–event timelines a semi-formal diagrammatic technique, introduced first to describe the behaviour of electronic mail receipt has also been successfully used to determine the appropriate auditory feedback for on-screen buttons. This was particularly impressive as the analysis showed (before experimentation) that the initial design idea would not solve the target problems, but also suggested an alternative design which future experiment confirmed to be effective [15, 6].

Expressing Status–Event Phenomena

While this status/event distinction may seem obvious, we might ask why it is useful for describing interface behaviour. Our operating assumption is that the distinction between status and event is natural in our understanding of interface behaviour. It follows, therefore, that the languages we use to specify an interface should reflect how we naturally think of them. That is not to say it is impossible to describe an interface without access to both status and event. On the contrary, most modern window-based interfaces are event-driven, which means that, at the lowest level, any and all interface behaviour must be described in terms of events. We can refer to the *grain*, or language bias, for such event-driven systems. The grain of an interface language refers to its natural tendency toward describing interface behaviour. As we have demonstrated above, some interface phenomena are more easily described in terms of events and some more easily in terms of status. Opening a window by clicking on an icon is easily described by a selection event usually linked to some mouse click whereas movement of the mouse cursor is easiest to think of as a time-varying status input. Some behaviour is a combination of both status and event. For example, selection of an item in a pull-down menu involves event input to reveal the menu, status input to wander up and down the menu and possibly reveal submenus, and event input to select an item from the menu.

If a language restricts expressions to events only, then status phenomena will be difficult, if not impossible, to express. When the natural expression of some interface behaviour goes against the grain of a specification language, then at least one of three things will occur:

- it will be specified incorrectly;
- it will be specified correctly but in a way that is difficult for readers of the specification to understand; or
- its specification will be ignored.

All three of these options are unfavourable. In this paper, we investigate the status/event biases of various specification languages used to describe interactive systems. We find that, despite the frequent occurrence of status/event phenomena, formal specification notations for interactive systems do not deal with both adequately. Notations have either an event only or status only grain.

This paper argues the advantages of using a formal model that is specifically designed to reflect both status and event behaviour within the interface. There are two main uses of such a formalism—as a high-level requirements language and as a low-level language for constructing widgets based on their intended behaviour.

In order to demonstrate these issues we build a formal model which can describe all kinds of status/event properties. However, the aim is not to sell the particular model and notation, but the general principal that both phenomena must be describable within an effective interface specification notation.

Structure

In the next section, we look at various models which have been used for interface specification. We show how these fail to uniformly treat status and event behaviour. Then in section 3, we use these simpler models as a guide to build up to the more complete model, which is an extension to the agent model developed by Abowd [2]. This presents both the model and a concrete syntax for the model.. Finally in sections 4 and 5 we look at examples of the use of the new model to describe single and multi-user interfaces. This exercise exposes several design problems which can be easily overlooked when using purely event based approaches.

2 Existing formal models

In this section we review various formal models and notations in the light of the status/event distinction. The models are characterised by which combinations of status/event input/output they support. Ignoring the case of a model with no input or output, there are 15 different combinations although not all this space is populated by existing systems.

In order to give a unified treatment, we take an agent-based perspective. Some interface models treat the entire system as a single entity, in which case this is the sole agent. Other models treat the system as an interacting network of agents. In addition, the user may be regarded as an agent. In each case, the terms input and output are used with respect to the agent being modelled.

Specification notations are used to characterise behaviour over time, and so a central feature of these models is how they characterise changes. This is often represented in terms of the changes to the states of agents. We take this perspective throughout, although it should be noted that for some notations the state is not explicit. We use the word state to refer to the *internal* information about the agent, whereas the status *output* is available outside the agent being modelled. It may be the case that the status output (where it exists) is some portion of the internal state, but this is not assumed.

There are three (non-trivial) *uniform* combinations where the input and output are of the same kind:

- event input – event output
- status input – status output
- event & status input – event & status output

Such combinations are, of course, particularly suitable for composition. That is a system can be built out of interacting agents where the outputs of some agents become the inputs of others. The asymmetric models tend to be where the system is regarded as a single agent and the asymmetry reflects the differences in the way the model treats the user's input to the system and the system's responses. However, this is not universally the case, and we will see that the Interactor model [18] is compositional but asymmetric (event input – event and status output).

Event Input – no Output

Perhaps the simplest model we can appeal to is one in which the state transitions are triggered by events from the environment, but where there is *no* output at all. At first this seems to be a silly and rather useless category. However, where the intention is not to capture the entire behaviour of a system it is quite reasonable. Indeed, various forms of grammars have long been used as a form of dialogue description, for example, Reisner's use of BNF [32] and Task Action Grammar (TAG) [33]. These grammars describe the possible sequences of user

input, but are often silent concerning system responses. Similarly some uses of State Transition Networks (STNs) deal only with the transitions between states caused by user inputs [25].

Such systems can be described by a simple state transition function:

$$new_state = doit(\ old_state,\ in_event\ )$$

As we have already mentioned grammars do not have an explicit state, but for purposes of comparison, they can be regarded as possessing an implicit state. Take for example the following BNF description of a mouse:

```
mouseup    ::=   Move  mouseup
               | Press mousedown
mousedown  ::=   Move    mousedown
               | Release mouseup
```

In this case, the non-terminals '`mouseup`' and '`mousedown`' can act as the states of the system. In a more complicated example more states may be required, but a (possibly infinite) state based representation is always possible (indeed many parsers work in precisely this fashion).

Event Input – Event Output

In a stimulus-response system, an input event leads to some state transition which results in output events. This is represented diagrammatically in Figure 1 and can be modelled using a modified state-transition function:

$$new_state, out_event_set = doit(\ old_state,\ in_event\ )$$

Many User Interface Management Systems [31] and User Interface Development Environments [19] use event-response systems or event based production-rules as their dialogue control component. In addition, most actual window managers are based on event loops, in which case, the output events are usually describing changes in the interface: update a window, create a dialogue box etc. Notice how the outputs to the window manager are principally status, even though they are described using events.

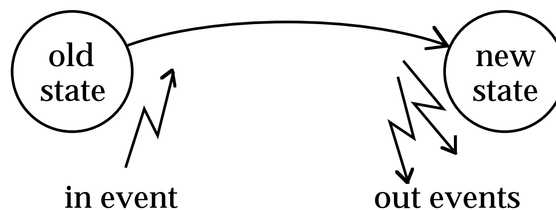


Figure 1: Event-in — Event-out

One of the important features of this class of notations is that they can use communication as a means of composing objects. The output events (or messages) of one object become inputs to some other objects in the system, which, in turn, trigger state transitions and further events. Thus a system can be described as a network of communicating agents. It is when agents are composed that the differences between the interpretation of events in different formalisms become apparent. In Agha's Actors model [4], communication is asynchronous. So that there may be an arbitrary delay between the sending of a message and its receipt. Furthermore, Actors does not even guarantee that the ordering of messages sent by one agent are preserved when received by another. In contrast, the process algebras, such as CSP (Communicating Sequential Processes [24]) and CCS (Calculus of Communicating Systems [26]), mainly adopt a synchronous model of communication and in CSP

the distinction between input events and output events is somewhat blurred when there is no explicit value passing. Also the bare process algebras, like grammars, leave the state implicit describing only the ordering of events.

As many interface issues are difficult to express without some model of state, some researchers [5, 1, 2, 35] have extended a process algebra with a state-based notation in order to make the state of the object more explicit in the specification. Since these notations adopted a CSP model for the process algebra, they all assume synchronous communication between objects. Also LOTOS has been used in the CNUCE Interactors model [29, 30] as it includes both a CCS-based process algebra to specify event order and an algebraic notation ACT-ONE to specify state values.

Many forms of state transition networks fall into this category too. Some annotate the arcs with the system's feedback, whereas others may add arcs which can indicate both input events and output events, as is done in Harel statecharts [22]. Likewise, where Petri nets have been adopted and extended for interface specification [], their event based is still evident.

As one would expect, the description of status-oriented behaviour is cumbersome using this class of notations. Look, for example, at the behaviour of a mouse dragging a box using CSP:

NoDrag	=		mouseDown {*while over a box*} → Dragging
		[]	mouseMove(x,y) → NoDrag
Dragging	=		mouseUp → NoDrag
			mouseMove(x,y) → moveBox(x,y) → Dragging

Notice how the connection between the mouse's position and that of the box has to be maintained by a sequence of little moves. Also notice the side condition 'while over a box'. This is not part of CSP, but has been included to capture the dependency on the mouse and box position. This side condition would be extremely difficult to express fully in CSP, or indeed even a notation where local state has been added as it relies on the internal state of two components of the system. Consider how much more difficult this would be if one wished to describe the way the wastebin highlights on an Apple Macintosh when an icon is dragged over it. We would need to discuss three status phenomena: the mouse position, the icon position and the wastebasket position.

When several status phenomena are linked, not only is an event description cumbersome, it also encourages over commitment. Consider the dragging of two selected items over a desktop. We might write:

Dragging	=	mouseUp → NoDrag
	[]	mouseMove(x,y) → moveObj1(x,y)
	[]	→ moveObj2(x,y) → Dragging

But, this says that the second object always lags behind the first. An implementation which moved the second first would be 'wrong' according to this specification. This bias can be overcome by using parallelism in the description, but all one really wants to say is that they are both at the same position as the mouse.

Status Input – Status Output

Dragging, the relationship between the mouse position and the position of the box, is a status–status mapping, and can be captured quite easily using standard software engineering notations, such as Z, VDM, or the algebraic specification methods. We can write something like:

$$dragging(box) \Rightarrow box.centre = mouse.position$$

In fact, the most complicated thing about this expression is the guard. It is not always true that the box is at the mouse position, only when it is being dragged. Of course, this leaves one with the awkward question of what the position is when the box is not being dragged! Unfortunately, this is rather more difficult as it is 'the *last* position it was dragged to'. To express this requires either an explicit model of time, or some event description.

Despite this shortcoming, pure status–status mappings can be very useful for expressing invariants – things which must *always* be true of a system. In particular, one can express the relationship between the display of

a system and its internal state. For example, in a word-processor it must always be the case that the screen display reflects faithfully the appropriate part of the document being edited. We can write this in Z:

Display

$document : \mathbb{N} \times \mathbb{N} \nrightarrow CHAR$
$screen : \mathbb{N} \times \mathbb{N} \nrightarrow CHAR$
$lines, columns : \mathbb{N}$ [size of screen]
$offset : \mathbb{N} \times \mathbb{N}$ [offset of screen image in document]

$\mathrm{dom}\, screen = (1..lines) \times (1..columns)$

$\forall (l, c) : \mathrm{dom}\, screen \bullet$
$\quad (l, c) + offset \in \mathrm{dom}\, document \Rightarrow$
$\qquad screen(l, c) = document((l, c) + offset)$
$\quad (l, c) + offset \notin \mathrm{dom}\, document \Rightarrow screen(l, c) = space$

Try writing that in CSP!

Notice also how easy it would be to add multiple displays at different offsets and hence begin to describe a group editor. Actually maintaining such an invariant between displays on machines spread over a network will, of course, be very difficult, but the specification can at least help one to see the aim of the system.

Event Input – Status Output

If one considers older computer systems, one notices that the input/output modalities are asymmetric. The dominant input device was the keyboard (event) whereas the output was the CRT display (status). It is not surprising that the PIE model [11, 12], an early formal model of interaction, inherited this bias (Figure 2).

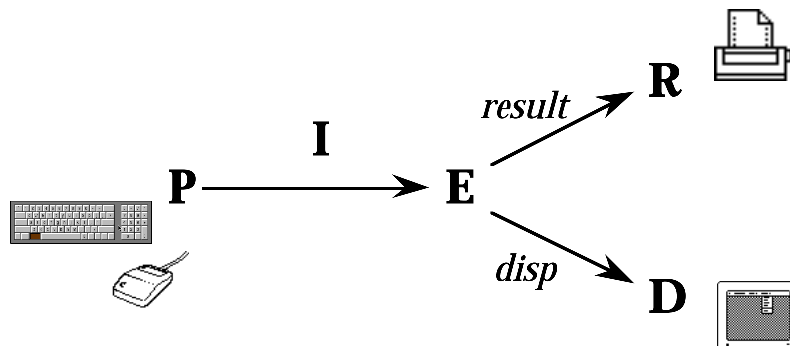


Figure 2: PIE model

The PIE model regards the inputs to be a sequence of user commands (for historic reasons the sequence of commands is called P). The command trace is interpreted by the system to give a current state (from the set E), from which the current display and the ultimate result of the system can be obtained. The PIE model can be regarded in terms of a state transition function and status–status mappings relating the current state and display.

state transition

$new_state = doit(old_state, command)$
$(I = doit^{\star})$

status–status map

$current_display = disp(state)$

If one regards mouse movement as a sequence of 'move' commands, this is a very general model of interaction. However, it is, our thesis that this would be an unnatural interpretation of mouse movement.

In addition to this special purpose model, the event-in/status-out representation has been used as the basis of many specifications of interactive systems using conventional notations (e.g., [34]). This may be achieved using functions to represent state update:

$$mouseDown(state, pos) = state'$$
where $\quad inside(pos, box) \Rightarrow state'.dragging = true$
$\quad\quad \ldots$

Or in the Z notation, 'delta' schemas may be used:

MouseDown
$\Delta State$
$pos? : \mathbb{N} \times \mathbb{N}$

$dragging' = (pos? \in box)$
$\ldots$

In both cases, the binding between function/schema names and the events they denote is informal.

Event Input – Event & Status Output

As we saw, the dominant form of output in interactive systems is status, the display, but an event-in/status-out model, like the PIE, is not compositional – one can talk about the system as a whole, but not about its construction. Duke and Harrison's Interactors [18] add a status output to an event based notation in order to be able to deal with display output. Similarly, Abowd's Agents [1, 2] have been augmented by *templates* describing those portions of the internal state which are available as output.

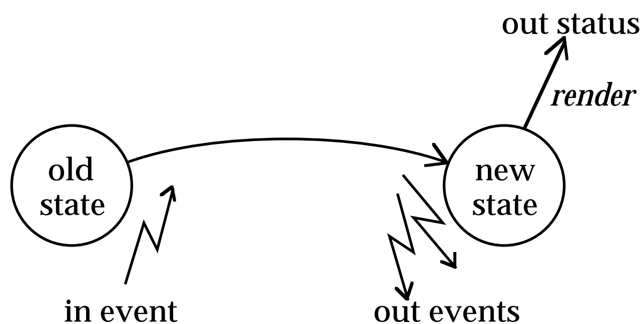


Figure 3: Event-in — Event & Status-out

Both these notations can be described using a state transition similar to that for the event-in/event-out notations augmented with a mapping from the state to the status output:

$$new_state, out_event_set = doit(\, old_state,\ in_event\,)$$
$$status_output = render(\, state\,)$$

Note the equation for '*render*' has been written as a function of simply '*state*'. This is to emphasise it is a relationship that always holds, not just at state transitions. In Duke and Harrison's work this is indicated by simply saying which state components are visible.

The resulting agents can communicate using events, just as with event-response systems. However, the status output is *not* available to other agents and is only for the purpose of describing the final user display.

This has two consequences. First, mouse input is still regarded as a sequence of move events. Second, one cannot easily layer a description where the internal layers have status output. For example, one might describe a display editor with the output being the screen display, then later want to use that description within a larger system where the display of the editor was just one window on a larger screen. In order to achieve this with an event-in/event&status-out system, one must describe the inner part of the system purely in terms of events. Only the very outermost layer has any status output at all.

Event & Status Input – Status Output

Probably the most serious problem with the PIE model for dealing with graphical systems was its inability to deal neatly with status input – in particular the mouse position. Because of this a variant of the PIE model was designed which could include mouse position (or other status) input [12].

In the modified PIE model, the state transition function depends not only on the current state and the command (the event which caused the transition) but also on the current mouse position (the status input):

$$new_state = doit(old_state, command, pos)$$

This means that it is easy to express conditions such as 'mouse is within the box' as they translate into a predicate about the current position and the current system state. However, an additional mechanism is needed to deal with dragging, the display function must also depend on the current mouse position:

$$current_display = disp(state, pos)$$

That is, output status is a function of state and input status.

This display function is an example of *interstitial* behaviour, as it describes what happens to the display between commands. Indeed, a lot of the dynamic behaviour of a graphical system is captured in this function.

In fact, this is only a limited form of interstitial behaviour as the status input only has a permanent effect when an event occurs. This behaviour has been termed *trajectory independent*. A drawing package requires a more complicated form of interstitial behaviour as the path that the mouse takes between events must be recorded during freehand drawing. The different forms of interstitial behaviour is a fascinating topic in itself which is partially investigated in [12][1], but initially we look only at the simple form of behaviour above.

Summary

We have now looked at six classes of model. Table 1 summarises the models and their capabilities. Ideally, we would like a model which captures all kinds of behaviour.

Before proposing such a model it is worth reiterating why it is necessary. Proponents of both event and status based approaches can justifiably argue that they can encompass all types of behaviour. For example, it is possible to model a status value in a process algebra:

```
StatusVar(x)  =      set(v) → StatusVar(v)
                 []  get(x) → StatusVar(x)
```

Indeed, this is similar how the user's display is handled in the CNUCE Interactors model [29, 30]. However, in their model an extra triggering event is added before their equivalent of the 'get(x)' event.

Similarly, it is possible to regard an event, such as the pressing of a mouse button as simply the change in a status – indeed, polling devices behave exactly like that.

The important thing is that, although both methods 'work' and indeed may even be the way the phenomena is implemented, they do not naturally represent the phenomena. As we discussed in the introduction, this leads to a specification which is constantly trying to go against the grain of the notation and thus almost certainly to one which is wrong.

[1] But not using the word interstitial!

	Event-Out	Event-In	Status-Out	Status-In
grammar		√		
stimulus response	√	√		
invariant in Z			√	√
PIE model		√	√	
Z Δ schema		√	√	?
interactors	√	√	√	
modified PIE		√	√	√
* Ideal model *	√	√	√	√

Table 1: Summary of models

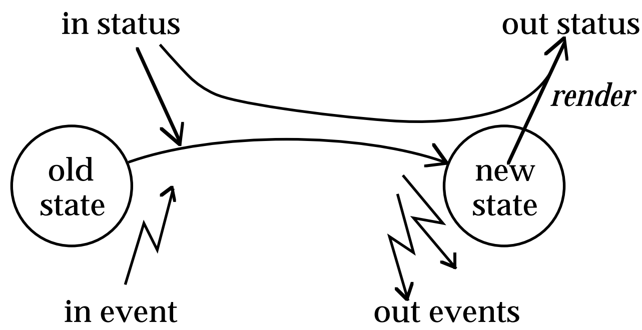


Figure 4: New model: Event & Status-in — Event & Status-out

3 Combining status and events – a new model

The final stage of our Odyssey, finding a model which fully encompasses both event and status behaviour is quite straightforward. We can simply take the relevant parts of the models in the preceding section and plug them together. Probably the most critical part is the interstitial behaviour inherited from the modified PIE model. To the modified PIE model's state transition function, we simply add event outputs as found in interactors or simple event-response systems:

state transition (action)
$new_state, out_event_set = doit(old_state, in_event, in_status)$

The interstitial behaviour is then a generalisation of the modified PIE's display function.

interstice (between actions)
$out_status = render(state, in_status)$

Remember that this represents the relationship which always hold true between actions. It can be used to capture continuous behaviour, for example, that the position of a slider follows that of the mouse, or (continuously true) discrete behaviour, for example, that the menu bar is highlighted when the option key is depressed.

Concrete notation

In order to deal with examples we need a concrete notation. However, we hold this notation lightly as we believe that it is the concepts which are important, not the particular notation. The notation used is in fact a little repetitive, and is really aimed at static configurations of agents. However, it serves to illustrate the utility of our approach.

For each agent, we list the names of the input events and the output events. We also list the names and types of each output event. For example, for a text editing agent this might look like:

EDITOR:

state:	$document : Doc_pos \nrightarrow Char$
	$cursor : Doc_pos$
in-events:	$keystrokes = \{$ a, b, c, ... $\}$
	mouse_up
	mouse_down
	mouse_click – for selecting text
in-status:	$offset_x : [0 \ldots 1]$ – from scroll-bar
	$offset_y : [0 \ldots 1]$
	$mouse_x : nat$ – from mouse
	$mouse_y : nat$
out-events:	*beep* – error signal
out-status:	$screen : Scr_pos \rightarrow Char$
	$scr_cursor : Scr_pos$

The types *Doc_pos* and *Scr_pos* are line and column positions in the document and screen respectively. Whereas the former ranges over $\mathbb{N} \times \mathbb{N}$, the latter only ranges over the valid screen coordinates.

We will also refer to some additional variables: *nos_lines*, *nos_cols*, *winpos*, *charht* and *charwid*, which are constant for the purpose of this example.

Not every agent has every type of input and output and indeed the above example has been somewhat contrived to get all kinds of event in one example. In contrast, the mouse agent has no inputs or state, because it directly represents the user's input to the system:

MOUSE:

state:	none
in-events:	none
in-status:	none
out-events:	*mouse_up*
	mouse_down
	mouse_click
out-status:	$mouse_x : nat$
	$mouse_y : nat$

The agent must then describe the possible state transitions. For each input event, there is a clause describing the changes to the state and also the output events which are raised. For example, the mouse click event for the editor might be as follows:

EDITOR – state transitions

on *mouse_click*:
  if $(mouse_x, mouse_y)$ over document window
  and $mouse_to_doc(mouse_x, mouse_y) \in \mathrm{dom}\, document$
    $cursor' = mouse_to_doc(mouse_x, mouse_y)$
    $document' = document$
  if $(mouse_x, mouse_y)$ over document window
  and $mouse_to_doc(mouse_x, mouse_y) \notin \mathrm{dom}\, document$
    raise beep

The function *mouse_to_doc* converts the mouse coordinates into a character position in the text. If this is over a valid character position in the document it is selected, otherwise the editor beeps.[2]

$$mouse_to_doc : Scr_pos \to Doc_pos$$

$$mouse_to_doc(x, y) = (line, col)$$
$$\text{where } line = offset_y \times nos_lines + \frac{y - winpos.y}{charht}$$
$$col = offset_x \times nos_cols + \frac{x - winpos.x}{charwid}$$

Finally, we must describe the interstitial behaviour. This defines the output status in terms of the input status and state of the agent:

EDITOR – interstitial behaviour

$$screen(x, y) = document(x + offset_x, y + offset_y) \quad \text{if } (x + offset_x, y + offset_y) \in \mathrm{dom}\, document$$
$$= \text{space} \quad \text{otherwise}$$

Notice that this involves both the state (*document*) and some input status (the offsets) in the calculation of the output status.

[2] A real editor would probably select the nearest legal position, but we want to demonstrate an event output!

More complex forms of interstitial behaviour

Although the interstitial behaviour defined above is sufficient for many purposes, more complex behaviour is possible. We have already mentioned the way that a drawing package may have trajectory dependent behaviour whilst the user is freehand drawing. To model this, the state transition and interstitial behaviour require not just the current input status, but the complete history of input status since the last event. If *Interval* is the set of possible intervals over some time domain, the semantic functions will be something like:

state transition
$new_state, out_event_set = doit(old_state, in_event, in_history)$

interstice
$out_status = render(state, in_history)$

The input history is a time series of input status over some interval:

$$\exists\, int \in Interval \textbf{ s.t. } in_history \in (int \rightarrow In_Status)$$

In the drawing example, the actual appearance depends only on the pixels over which the mouse passed (ran *in_history*). However, more complex examples may require the complete time-series. For instance, if one uses continuous time, one is able to describe so called ‘hybrid systems’ where discrete (digital) behaviour is mixed with continuous (analogue) [20]. This is used to model control of physical processes where the control device is digital, but the physical process is continuous. In such cases the interstitial behaviour is usually limited to differential or integral equations. These can be useful in interface specification also. For example, if an analogue joystick were used as an input device, then the position of the cursor on the screen would move with velocity determined by the joystick. This could be expressed:

$$pos_x = \int joy_x$$

The integral can be thought of as just one special form of function over the status input history.

In fact, the various notations developed in the study of hybrid systems are closest to the concerns of this paper. For example, the Extended Duration Calculus [8] has mechanisms to deal with continuous values and differential equations during interstitial periods as well the ability to reason about values of predicates over intervals inherited from the Duration Calculus itself [7]. The Duration Calculus also has the notion of an integral of a predicate, which is another kind of interstitial operator. It is effectively the normal integral of the characteristic function of the predicate and yields the amount of time the predicate is true during the interval.

Sometimes, the only important thing is when the status input crosses significant barriers. That is when there is a status-change event. This can be dealt with as a form of trajectory dependent interstitial behaviour, but if the status change is perceived as being an event for the user, it is better to represent it as such. One way of representing this is by allowing predicates as part of the state transition behaviour, not as guards, but as event generators. For example, in the case of the editor agent we might want to add a condition of the form:

on $\uparrow (cursor \notin screen_window)$:
move offset to re-center cursor within window

The notation $\uparrow P$ is read as “when the predicate P becomes true”. Similarly one can write $\downarrow P$ – “when the predicate P becomes false”. The event would occur, for instance, when the user was typing and the cursor got to the end of the screen. Note that this condition would often be combined with additional guards. If the user explicitly scrolls to the new location using the scroll bar then the corrective action would not be required.

This up-arrow/down-arrow notation is used in several formalisms to refer to the set of times when a predicate becomes true/false. For example, Moffet *et al.* use it as part of a requirements capture model [27].

Another form of event, which can be regarded as a special form of status-change event, is a timed event. Again, one could introduce special syntax for this.

These different forms of interstitial behaviour are all useful in different circumstances, but, we will use the simpler, trajectory independent, form in the rest of this paper. Our aim is to show that mixed status/event descriptions are necessary to adequately describe the user interface, and even restricting ourselves to this simple form we will find important issues are raised.

4 Example – slider control

We now consider the complete specification of a slider control as used in many graphical user interfaces. The appearance of the slider is shown in Figure 5. The user has two ways of manipulating the slider control.

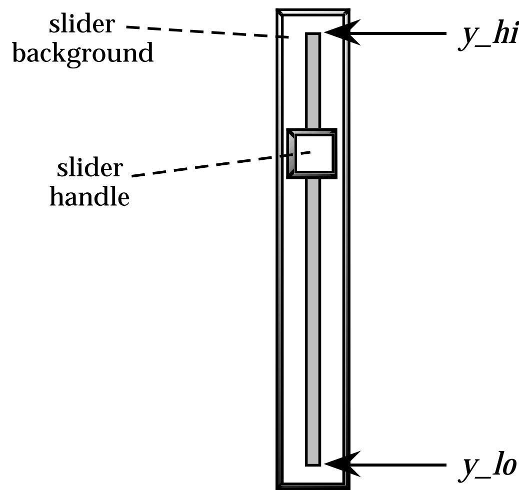


Figure 5: Slider control

- It may be dragged by the mouse. While the mouse is held down over the scroll-bar, the slider's handle moves up and down with the mouse. If the mouse is released within the scroll area, the handle moves to the relevant position. If however the mouse is released outside the scroll area, the slider snaps back to its original position.
- If the mouse is clicked over the scroll area, the handle moves to that position.

We do not consider forms of jump scrolling, such as screen-by-screen or line-by-line scrolling. These would make the description longer, but not fundamentally more complicated. Indeed, the necessary information is another example of status input, coming from the application using the scroll-bar. That is, the transformation of status is a two-way affair! Similarly, many scroll-bars show the position of the handle as an outline whilst dragging, we will not put this in the initial version.

A more difficult issue, is the fact that a real scroll-bar has mixed control. That is, the position of the scrolling may be changed both by manipulating the scroll-bar directly, but also by typing in the document which may move the window indirectly. We will return to this issue later in Section 5.

We will consider the components of the slider specification in the same order as those of the editor. However, this specification will be complete, albeit simplified.

State and Signature

First, we look at the state and inputs and outputs of the slider:

SLIDER:

state:	*save_val* : [0 . . . 1]
	dragging : *bool*
	delta: N
in-events:	*mouse_up*, *mouse_down*, *mouse_click*
in-status:	*mouse_x*, *mouse_y*
out-events:	none – could have change event
out-status:	*slider_val* : [0 . . . 1]
	save_val – from state
	dragging

The inputs all come from the MOUSE agent. The state variable '*save_val*' is used to hold the original position of the slider whilst it is being dragged and also to record the current position of the slider between dragging. The dragging variable records when the user is engaged in dragging. Note that this *cannot* be inferred from the state of the mouse buttons and the position of the mouse as the user may Note that both these state variable are also made directly available as output status. The last component of the state is used to record how far the mouse was from the centre of the handle when dragging began.

The three output status can be used by the application to set its offset in the document, and by the display manager to position the handle on the screen. If an outline of the handle is dragged around, the mapping between the status outputs and the screen display would be as follows:

graphic	where	when
handle	*save_val*	always
outline	*slider_val*	*dragging* = *true*
	not displayed	*dragging* = *false*

Alternatively, one might wish to move the actual handle around in which case the handle would always be at *slider_val*. Note that these are simply two options for the status–status mapping between the abstract slider state and the display. Similarly, we can choose whether to attach the offset in the document to *save_val* – in which case the text would only scroll when dragging was complete, or to *slider_val* when the text would scroll up and down as the user moves the mouse.

4.1 State transitions

We next look at the behaviour of the slider when events occur. Because we are able to model interstitial behaviour, we do not consider mouse movement an event. So, we only need to examine the effect of mouse button events.

SLIDER – state transitions

on *mouse_down*: (start to drag)
  if (*mouse_x*, *mouse_y*) over slider handle
    $dragging' = true$
    $save_val' = save_val$
    $delta' = mouse_y - calc_pos(save_val)$

on *mouse_up*: (finish drag)
  $dragging' = false$
  if dragging and (*mouse_x*, *mouse_y*) over slider background
    $save_val' = calc_val(mouse_y)$
  else $save_val' = save_val$

on $mouse_click$: (jump scroll)

$dragging' = false$

if $(mouse_x, mouse_y)$ over slider background

$save_val' = calc_val(mouse_y)$

else $save_val' = save_val$

The functions *calc_val* and *calc_pos* translate back and forth between slider values (between 0 and 1) and screen positions. They are not inverses of one another, as one of them uses *delta* to account for the mouse not 'holding' the handle in the middle.

$$calc_pos : [0 \ldots 1] \to \mathbb{N}$$
$$calc_val : \mathbb{N} \to [0 \ldots 1]$$
$$calc_pos(v) = v \times (y_hi - y_lo) + y_lo$$
$$calc_val(y) = \frac{y - delta - y_lo}{y_hi - y_lo}$$

Note how each transition depends both on the internal state of the slider agent and the status input (mouse position). In addition, the state variables *delta* and *dragging* depend in quite a complicated way on the history of user actions. It would be quite hard to describe this without using explicit state.

Interstice

Finally, we look at the interstitial behaviour of the slider. This is very important as it used to generate the constant feedback which is obviously necessary for the usability of the control. There is no need to specify the values of the *dragging* and *save_val* output status as these are taken directly from the corresponding state variables. Only *slider_val* needs to be specified:

SLIDER – interstitial behaviour

$slider_val = calc_val(mouse_y)$

when dragging and

$(mouse_x, mouse_y)$ over slider background

$= save_val$ otherwise

The critical thing to note is that this depends on both the status input (mouse position) and the current state (*dragging* and *delta*). We can not simplify the formula to:

$$slider_val = calc_val(mouse_y) \text{when mouse over the slider background}$$

Consider what would happen if the user depressed the mouse outside the slider (say over the screen background) and then dragged the mouse over the slider. With the simpler formula, the slider would begin to operate, whereas we only want it to follow the mouse position when the mouse was originally depressed over the slider handle. This dependence on history is captured by the state variable *dragging*.

We need everything

The example, was carefully chosen so as to demonstrate that all the kinds of mappings we have discussed are indeed necessary. The state transitions depended on what input event had occurred, the current state and the status input. Also the status output was a function of both the state and the status input. Furthermore, without the status output and the associated interstitial behaviour, we would not have been able to capture the user feedback.

5 Mixed control and groupware

As we noted earlier, the description of the scroll-bar is a little unrealistic as it assumes that user interaction with the widget is the only way the scroll position will change. This is probably acceptable for a drawing package, where the user must explicitly move the window when the point of interest is off screen However, in a word-processor one expects that the window will move as one types. For example, imagine you have a 4000 word document with the scroll window and cursor at the beginning. After a few hours you have typed another 4000 words, but not explicitly moved the scroll-bar. You would expect the window to now be in the middle of the document, not still at the beginning!

In fact, while you are typing, the scroll-bar effectively moves with virtually every character typed (although it will only perceptible move when the change exceeds a pixel). This is because the scroll-bar represents the proportion of the way the window is into the document, whereas normally the beginning of the window is kept fixed. Thus in a small document, the scroll-bar will move slightly *upwards* as the user types. The more dramatic change occurs when the typing would take the insertion point of the bottom of the screen, at which point the screen usually scrolls to a new position, often centring the insertion point in the screen, or moving it to the top line. This will move the scroll-bar downwards. That is, as you type the scroll-bar's position takes a saw-tooth path jiggling a little back and then jumping forward as it gradually makes it's way downwards.

In a single-user interface, the two modes of interaction (typing and scroll-bar dragging) rarely happen concurrently, and so some ad hoc fix can be applied when they do. For example, in the Macintosh interface, typing is buffered while the scroll-bar is being manipulated.

Modified slider

In order to incorporate this sort of behaviour, the slider agent needs to be changed somewhat, either adding events (from the application) to update the slider position, or making the slider position an *input* status. If the second option were adopted, the slider would emit a 'changed' event when the user finished scrolling and the application would be responsible for moving the actual offset to match the one the user set. In this case, the handle is actually a display device whereas the outlined box is the real input.

The new signature of the slider would be as follows:

SHARED_SLIDER:

state:	*dragging* : *bool*
	delta: N
in-events:	*mouse_up*, *mouse_down*, *mouse_click*
in-status:	*mouse_x*, *mouse_y*
	save_val : [0 . . . 1]
out-events:	*changed*(*v* : [0 . . . 1])
out-status:	*slider_val* : [0 . . . 1]
	save_val – from input status
	dragging – from input state

Obviously all the places in the specification where *save_val* is assigned a value would need to be removed and in addition, we need to generate changed events at the appropriate places. For example, the state transition for the mouse up event would become:

on *mouse_up*: (finish drag)
 $dragging' = false$
 if dragging and (*mouse_x*, *mouse_y*)over slider background
 raise *changed*(*calc_val*(*mouse_y*))

The *changed* event has been given a parameter value, this could also have been picked up from an appropriate output status (*slider_val*). However, the value is directly about the event so it seems more natural to include it with the event.

Note that, the changes introduced are not an implementation fix, but reflect a different conceptual model of the slider's operation. The fact that the original (and deliberately simplified) model could not describe the situation pushed us to think again and more clearly about the role of the slider in the interaction.

If the slider had been described entirely in terms of events (as would happen in a typical window manager), then the above problem would never have surfaced. There would have been an ad hoc solution which would have *emerged* from the particular event orders, but there would have been no need to explicitly *design* this behaviour. It is not clear whether the behaviour on a particular platform (for example the buffering on the Macintosh) is due to chance or design. However, it is clear that a formalism ought to force interface developers to face these critical design decisions.

Groupware problems

Ad hoc methods may work in a single-user application, but are unlikely to be satisfactory in a multi-user interface where one user may be typing whilst another is scrolling. In a tightly coupled system (with shared cursors and shared scroll position), one could simply lock the text whilst a slider is being dragged. The other users' typing could then be buffered just like the single-user case. However, in loosely coupled interaction, where the users may be typing in different parts of the document this would be unacceptable – periodically, and inexplicably, each user would find that the system would freeze momentarily (while the other user was scrolling), and then, just as suddenly, would spit out their buffered typing! On the contrary, users of a loosely coupled system expect to be able to continue typing whilst another user is scrolling.

This really forces the issue about where the control of the text scroll position lies. If each user has a different scroll position, then we need to consider the users viewport into the document as a separate agent. The document itself is conceptually shared by all the users. It has even more complicated control issues!

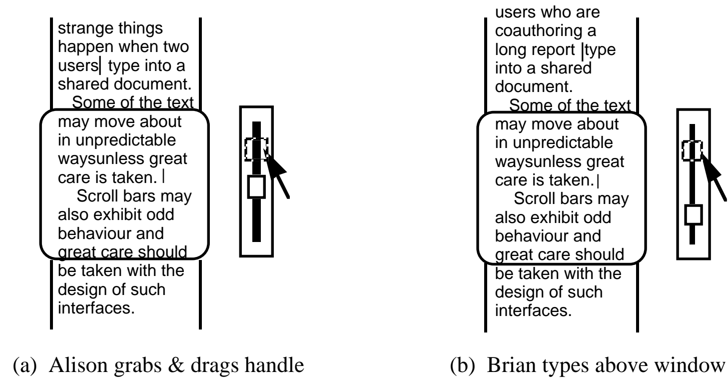
If we use the second slider definition, with the value as status input, it is easy to specify sensible shared activity. If the user or any other user types, then the slider position will move accordingly. For example, if someone is typing above a user's window then that user's slider will move down (as the offset into the text increases). When the user grabs the slider with the mouse, the outline of the handle is dragged with the mouse. However, the handle itself can still move up or down as other users type.

Figure 6 shows a typical scenario. Alison and Brian are editing a document together. The figure shows Alison's slider and window onto the shared text. The text just above and below Alison's window is also shown and Brian's insertion point is in the part of the document above Alison's window. Alison grabs the slider's handle with her mouse and starts to drag the outline handle upwards. While she is positioning the slider Brian types. Because there is now more text above Alison's window, the handle of her slider (but not the outline) moves downwards. Finally, Alison releases her mouse button and her slider handle and window move to the new position.

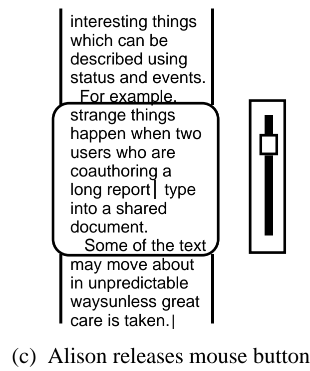
We cannot put the full definition of all the agents involved in this here, but the basic architecture is shown in Figure 7. The straight arrows denote status links and the jagged arrows are events. Each user has their own viewport and associated slider. The document is a shared agent and each user's window is derived by a status–status mapping from the current state of the document and the current offset for that user. Of course, this only represents one possible shared editor configuration: each user could be allowed more than one window into the document, or we could explicitly model a replicated document (especially as we refine the specification into an implementable system).

Other shared values and dynamic pointers

In fact, the slider is a simple example of the general problem of shared values with mixed control. The shared document is another, more complicated example. The appropriate use of status–status mappings simplifies the



(a) Alison grabs & drags handle (b) Brian types above window



(c) Alison releases mouse button

Figure 6: Shared use of slider

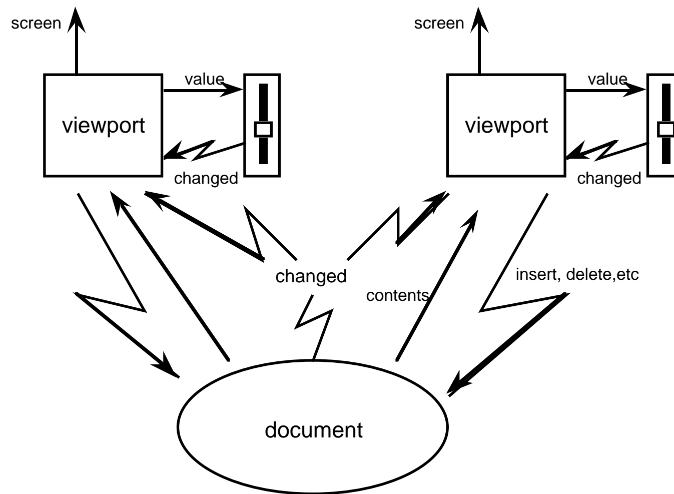


Figure 7: Architecture of shared editor

display part of the system and in so doing allows us to focus on the necessarily complex issues of shared update. In the example in Figure 6, Brian inserts text above Alison's window. This has two effects:

content – The contents of the document change and so any window showing that portion of the text ought to reflect the change.

position – As Brian types the position of Alison's window and insertion point move correspondingly.

The first of these is described easily by the status–status mappings so we concentrate on the latter. The update of pointers into sequential text is a general problem which arises in several contexts and has been addressed previously using the concept of *dynamic pointers* [12, 16]. Dynamic pointers are an area of study in their own right, so only a flavour can be given here. Basically, any object which can be updated should be able to support pointers which 'move' as the object changes, so that they always point to the 'same' (semantic) position. In text with insertion and deletion operations these mean that pointers after the insertion point are automatically shifted forward or backward respectively.

In practice, for a shared editor, this means one of the following:

- The centralised document agent 'knows' about all the users of the document and owns their insertion points and window offsets. That is, virtually all of the functionality is centralised.
- The shared document agent informs each user's viewport about changes and the user's viewports perform the appropriate updates to their pointers. That is the viewports own their insertion points and window offsets.
- The viewports describe their offsets and insertion points in terms of abstract pointer objects. Each viewport must 'ask' the shared document when it wants to convert one of these pointers into an absolute text position (i.e., offset is at line 33). These pointer objects are owned and updated by the central document agent, but it does not know what these pointers represent.

The first option is reasonable for simple systems, but leads to an amorphous, and hence hard to analyse, design. Either of the second or third options serve to modularise the specification. The third is probably the best at an abstract design level as the viewports do not need to know about the different types of document changes which can occur. Indeed, this third option leads to an elegant specification of the desired properties, which would not be possible without both the concepts of dynamic pointers and status/event description. Furthermore, it is readily transformed into an implementable design.

Are problems a problem

This section has been dealing with the complex issue of mixed control of objects in interfaces (both single and multi-user). The use of status/event descriptions have not given us a pat solution to these problems. Indeed, it would be worrying of they did as this is a fundamentally difficult area and there is probably no single right answer. Instead, the appropriate use of status and event description has allowed us to focus on the key issues and has exposed the nature of the problem. Contrast this again with a purely event based description, as would arise if one implemented using a standard toolkit, or event an event based specification notation. In this case, the behaviour would depend the order in which various events arrived. Even if the designer never considered the problem of mixed control a solution would emerge. However, this solution would have never been designed – it may be good, or it may be awful. With a status/event description the designer must face these issues.

6 Refinement and implementation

This chapter is concerned principally with the effective description and specification of interactive systems. However, any design must eventually be implemented and so we briefly describe some of the issues relating to the refinement of a status/event based description into a running system.

time	action	a	x	y	z	
0		3	6	9	3	
1	change to a	7	6	9	3	
2	update x	7	14	9	3	
3	update z	7	14	9	-5	* invariant broken
4	update y	7	14	21	-5	
5	update z	7	14	21	7	

Table 2: Update order x-z-y-z

Throughout, we have stressed how important it is to describe event and status behaviour appropriately. However, it can be argued that on a digital computer everything is ultimately event driven. This poses no problem for state transitions as any input status upon which the transition depends can be sampled when the event occurs. Status–status mappings are obviously more problematical. In practice, status–status mappings are mediated by events, either demand driven or data driven. In fact, the opposite is also the case and agents often use status to mediate events: e.g., the setting of shared variables, use of file system or databases for interprocess communication, but we will focus here on the status–status problems.

The maintenance of status–status mappings has both behavioural and coding implications.

Behavioural issues

Mediation of status–status mappings by events can lead to, at best, delays and, at worst, inconsistency. The former, is clear, but perhaps the latter requires an example. Imagine we have a system with four status values a, x, y and z. The first, a, is obtained from outside the system and the remainder are connected by status–status mappings to a. The mappings the system seeks to maintain are:

$$\begin{aligned} x &= 2 \times a \\ y &= 3 \times a \\ z &= y - x \end{aligned}$$

Note that one can therefore infer that if a is always positive then z will also always be positive (in fact equal to a).

Imagine that the system starts off in the consistent state:

$$\begin{aligned} a &= 3 \\ x &= 6 \\ y &= 9 \\ z &= 3 \end{aligned}$$

The external status, a, then is updated to become 7. The system has to repair the mappings in some way. Depending on the way the system works this might happen in a variety of ways. If the updates happen in the order x-y-z or y-x-z, the different variables are temporarily inconsistent with one another, but at any moment each is consistent with one or other state of a. However, if the system were to update the system in the order x-z-y-z, then the intermediate value of z would be -5 (see Table 2) – a negative value.

Unfortunately, the above update order could easily arise in certain types of constraint maintenance systems. The update to a causes the first two constraints to become invalidated. The system chooses the former to deal with, resulting in the update to x. This then invalidates the third constraint. If the system is working in a depth

first fashion this means that the third constraint will be repaired next updating z. Only then would the second constraint be addressed updating y which would again invalidate the last constraint and so finally z would be updated a second time.

Obviously, a more complex system might be able to detect such sequences, but even then there is the likelihood that the user will be presented with inconsistent data from several sources. If the user interface updates were delayed until a consistent state were reached then the latency could become excessive. Furthermore, in a distributed environment synchronised update is impossible.

These problems with status–status mappings should be addressed explicitly at some well defined place in the design process. Again, the use of purely event based notations from the outset would hide, but not solve, these problems.

We have explored these mediation and refinement issues in greater depth elsewhere [17], but more work is certainly required in this area.

Coding considerations

One approach to implementation is to 'refine' the status–status mappings of the interstitial behaviour into equivalent data or demand-driven event based behaviour. The resulting event-based description can then be implemented using standard programming techniques and interface toolkits. The word 'refine' is used guardedly as, for the reasons addressed above, the resulting event based description will not have identical behaviour to the original status/event description. Some weakening of temporal and semantic properties is inevitable.

Alternatively, some user-interface toolkits are based on some form of constraint system, for example, Garnet [28] and Rendezvous [23]. This will maintain status–status mappings on behalf of the developer, although one should be aware that they do not guarantee immunity from the problems of mediated mappings as was demonstrated above.

Active variables as used in Suite [9, 10] are also a way of supporting status–status mappings by automatically generating appropriate change events. Furthermore, Suite supports shared variables over distributed platforms. Of course, it does not absolve the designer from considering the issues of mixed control of shared variables – Suite implements particular policies for dealing with these issues and the designer must decide whether these are appropriate and also select various parameters to tune the policies to a particular system.

In a similar vein, the authors have noted that event based implementations of status–status mappings depend on broadcasting change events to other interested agents (so that they can repair the mappings). They have therefore been experimenting with an implementation paradigm which directly supports this type of event (triggers) allowing interested agents to register call-backs for triggers generated by other agents. This is in addition to more standard message type events. Like Suite this relieves the groupware developer of low-level network and distributed system coding.

7 Summary

The status/event description is necessary for the natural and effective description of user interfaces. Different interface specification techniques can be classified depending on how they deal with status and events, but none deal with both status and event uniformly for input and output. We have shown that it is possible to define a specification technique which embodies both status and events and have looked at examples of the use of such an approach. We are not committed to the particular concrete syntax used during the chapter and would be happy to see other styles of notation extended in a similar fashion. The two crucial features which would be in any such notation are definitions of both event induced state transitions and interstitial behaviour. Indeed, it is the interstitial behaviour, the fluid change *between* actions, which is largely responsible for the sense of responsiveness in the interface [13]. The use of status/event descriptions exposes several issues which need to be addressed in many interfaces, especially in the design of groupware.

References

[1] G. D. Abowd. Agents: communicating interactive processes. In *Human–Computer Interaction – INTERACT'90*, pages 143–148. Elsevier Science Publishers, 1990.

[2] G. D. Abowd. *Formal Aspects of Human-Computer Interaction*. D.Phil. thesis, Oxford University, Programming Research Group, 1991.

[3] G. D. Abowd and A. J. Dix. Integrating status and event phenomena in formal specifications of interactive systems. In *SIGSOFT'94*, D. Wile (ed.), pages 44–52. ACM Press, 1994.

[4] G. A. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1986.

[5] H. Alexander. Executable specifications as an aid to dialogue design. In *Human–Computer Interaction – INTERACT'87*, pages 739–744. North Holland, 1987.

[6] . S. A. Brewster, P. C. Wright, A. J. Dix and A. D. N. Edwards. The Sonic Enhancement of Graphical Buttons. In *Human–Computer Interaction – Interact'95*, Lillehammer, Norway. pages 43–48, 1990.

[7] Z. Chaochen, C. A. R. Hoare and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5), pages 269–276, 1991.

[8] Z. Chaochen, A. P. Ravn and M. R. Hansen. An extended duration calculus for hybrid real-time systems in *Hybrid Systems*, R. L. Grossman, A. Nerode, A. P. Ravn and H. Rischel (eds.), LNCS 736, pages 36–59. Springer-Verlag, 1993.

[9] P. Dewan. A tour of the Suite user interface software. In *UIST'90: Proceedings of the 3rd ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 57–65. ACM, 1990.

[10] P. Dewan and R. Choudhary. A high-level ,and flexible framework for implementing mulituser interfaces. *ACM Transaction on Information Systems*, 10(4), pages 345–380, October 1992.

[11] A. J. Dix and C. Runciman. Abstract models of interactive systems. In *People and Computers: Designing the Interface – HCI'87*, pages 13–22. Cambridge University Press, 1987.

[12] A. J. Dix. *Formal Methods for Interactive Systems*. Academic Press, 1991.

[13] A. J. Dix. Status and events: static and dynamic properties of interactive systems. *Proceedings of the Eurographics Seminar: Formal Methods in Computer Graphics*, D. A. Duce (ed.). Marina di Carrara, Italy, 1991.

[14] A. Dix, J. Finlay, G. Abowd, and R. Beale. *Human–Computer Interaction*. Prentice Hall International, 1993. ISBN 0-13-458266-7 (hardback) 0-13-437211-5 (paperback).

[15] . A. J. Dix and S. A. Brewster. Causing Trouble with Buttons. In *Ancillary Proceedings of HCI'94*, Glasgow, UK, 1994.

[16] A. J. Dix. Dynamic pointers and threads. *Collaborative Computing*, 1(3), pages 191–216, 1995.

[17] A. J. Dix and G. Abowd. Delays and temporal incoherence due to mediated status–status mappings. *SIGCHI Bullitin*, April 1996.

[18] D. J. Duke and M. D. Harrison. Abstract interaction objects. *Computer Graphics Forum*, 12(3), pages 25–36, 1993.

[19] C. Gram and G. Cockton (editors). *Design Principles for Interactive Software*, Chapman & Hall, 1996.

[20] Robert L. Grossman, Anil Nerode, Anders P. Ravn and Hans Rischel (editors). *Hybrid Systems*, LNCS 736, Springer-Verlag, 1993.

[21] J. Grudin. Why CSCW application fail: Problems in the design and evaluation of organizational interfaces. In *CSCW'88: Proceedings of the Conference on Computer Supported Cooperative Work*, pages 85–94. ACM, 1988.

[22] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3), pages 231–274, 1987.

[23] Ralph D. Hill. The Rendezvous constraint management system. In *UIST'93: Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 225–234. ACM, 1993.

[24] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[25] R. J. K. Jacob. Using formal specification in the design of a human–computer interface. *Communications of the ACM*, 26(4), pages 259–264, 1983.

[26] R. Milner. *A Calculus of Communicating Systems*. LNCS 92, pages 36–59. Springer-Verlag, 1980.

[27] J. Moffet, J. Hall, A, Coombes and J. McDermid. A model for a casual logic for requirements engineering. *Requirements Engineering*, 1(1), pages 27–46, 1996.

[28] B. A. Myers, D. A. Guise, R. B. Dannenberg, B. Vander Zanden, D. S. Kosbie, E. Pervin, A. Mickish and P. Marchal. Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces. *IEEE Computer*, 23(11), pages 71–85, 1990.

[29] F. Paternó and G. Faconti On the LOTOS use to describe graphical interaction. In *HCI'92: People and Computers VII*, pages 155–53. Cambridge University Press, 1992.

[30] F. Paternó, M. S. Sciacchitano and J. Lowgren A user interface evaluation mapping physical user actions to task-driven formal specifications. In *Design, Specification and Verification of Interactive Systems '95*, P. Palanque and R. Bastide (eds.), pages 155–173. Springer-Verlag, 1995.

[31] G. Pfaff (editor). *User Interface Management Systems*, Springer-Verlag, 1985.

[32] P. Reisner. Forma grammar and human factors design of an interactive system. *IEEE Transactions on Software Engineering*, SE-7(2), pages 229–240, 1981.

[33] F. Schiele and T. Green. HCI formalisms and cognitive psychology: the case of task-action grammars. Chapter 2 in *Formal Methods in Human–Computer Interaction*, M. D. Harrison and H. W. Thimbleby (eds.), Cambridge University Press, 1990.

[34] B. Sufrin. Formal specification of a display editor. *Science of Computer Programming*, 1, pages 157–202, 1982.

[35] B. Sufrin and J. He. Specification, refinemnet and analysis of interactive processes. Chapter 9 in *Formal Methods in Human–Computer Interaction*, M. D. Harrison and H. W. Thimbleby (eds.), Cambridge University Press, 1990.

Words

We have adopted a rather archaic use of the term interstice, but it is a definition which captures the essence of what we are after:

> **Interstice** . . . **2.** An intervening space of time; an interval between actions. (Shorter Oxford English Dictionary, Third Edition)

Arguably, the word has connotations which suggest small, often vanishingly small gaps which is in contrast to our assertion that much of the activity in a system happens during the interstices. However, this discordant note perhaps serves to emphasise our point: many interface descriptions do concentrate solely on actions and overlook the activity during the interstice. We seek to promote the often overlooked interstitial behaviour and put it on equal footing with the behaviour at actions. The first step in such a process is of course to give this behaviour a name.