Integrating status and event phenomena in formal specifications of interactive systems

Gregory D. Abowd
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
USA

Alan J. Dix
School of Computing and Mathematics
The University of Huddersfield
Queensgate, Huddersfield HD1 3DH
United Kingdom

Abstract

In this paper we investigate the appropriateness of formal specification languages for the description of user interface phenomena. Specifically, we are concerned with the distinction between continuously available information, which we call status, and atomic, non-persistent information, which we call events. We propose a hybrid model and notation to address status and event phenomena symmetrically. We demonstrate the effectiveness of this model for designing and understanding mixed control interaction, an especially important topic in the design of multi-user systems.

Keywords: formal specification, interactive system design, multi-user systems, mixed control interaction

1 Introduction

Implementing and reasoning about the design of any interactive system is a difficult and error-prone activity. A common problem is that the behavior of the system under certain circumstances is not correct from the user's perspective and this incorrect behavior ultimately destroys the user's confidence in the system. One part of the problem is that the languages used to specify, design and implement the system do not enable a perspective that matches that of the users. To specify and reason about the properties of the system, we need languages that more naturally express the concepts of interest in that system. In particular, we are interested in formal specification languages that provide abstract descriptions of the system early on in development when critical design decisions are made.

In this paper, we distinguish between *status* and *event* phenomena that occur at the user interface. The dichotomy between status and event phenomena has been pointed out by the authors previously [9, 10]. 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 signaling 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.

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 behavior 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.

While this status/event distinction may seem obvious, we might ask why it is useful for describing interface behavior. Our operating assumption is that the distinction between status and event is a natural one for understanding interface behavior. 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 any and all interface behavior must be described in terms of events. It is our belief, however, that the bias of these programming languages toward event-only description restricts them from due consideration of status phenomena.

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

- it will be described incorrectly;
- it will be described correctly but in a way that is difficult for others to understand; or
- its description will be ignored.

All three of these options are unfavorable.

To appear in the Proceedings of the ACM SIGSOFT'94 Symposium on Foundations of Software Engineering, December, 1994, New Orleans, Louisiana.

Overview

In Section 2, we will briefly overview the range of formal notations that have been used to specify interactive systems, providing a classification based on how these notations treat both status and event phenomena. This classification demonstrates that there is not a specification notation that treats status and event information symmetrically. This motivates our development in Section 3 of a prototype hybrid model and notation that provides equal treatment to the description of status and event. We will provide some small examples to demonstrate how the notation can be used to provide more natural specifications of interactive widgets in single- and multi-user applications with mixed control dialogs. In Section 4, we will provide a more complicated specification from the domain of groupware to demonstrate how the status/event language forces consideration of relevant design decisions, rather than leaving them for the user to discover. In Section 5, we will relate our work on specification languages to implementation mechanisms in interactive system builders and programming languages.

2 A classification of specification approaches for interactive systems

A number of researchers have used formal specification notations to describe and analyze interactive systems (for reviews of this work, see [2, 3, 15]). In this section, we provide a slightly different classification of those approaches in terms of how they handle the description of both status and event information. All of these approaches treat an interactive system as reactive; the entities they describe receive input, transform their internal state based on that input, and produce some output. The internal state[1] is either explicitly described or remains implicit. For any given specification approach, we determine whether status (continually varying information) or events (atomic actions), or both are used to describe the input and output behavior.

Figure 1 provides a graphical overview of our classification scheme. All approaches assume that entities have an internal state (implicit or explicit) that evolves by means of state transitions. The state transition can be triggered by an input event (such as a mouse click or keystroke) and can result in a set of events being announced after the state transition. Continuously available status information (such as mouse position, or position of the handle in a scrollbar) can be used as input to define a state transition or can be altered as the result of a state transition and rendered as output to the rest of the system.

Elsewhere, we have fully explored the space of formal specification languages applied to interactive systems and user interfaces with respect to the status/event distinction [8]. We summarize the results of this classification in Table 1. Where possible, we have included references to work that specifically relate to the specification of interactive systems or user interfaces.

There are a couple of important points to make in reference to Table 1. We can ask about the compositionality provided by any specification approach. For composition to work, the approach must be symmetric with respect to either status or events, or both. In other words, in order to build larger entities from smaller entities, the approach must provide a way to compose input and output and this means that if events are used for input then they must also be used for output. Approaches that are not symmetric in this way will not support composition; specifications using these languages will not be very modular.

The inclusion of status information as input allows the description of state transitions that depend on continuous information (e.g., the mouse position). Status input and output together allow the specification of continous relationships that are a frequent and important occurrence in modern graphical user interfaces. For example, to describe dragging in a direct manipulation interface, we require a continual relationship between the mouse cursor (input status) and an icon or image (the output status). These status–status relationships only hold between significant events at the interface. For example, dragging holds between the selection of an object and its eventual release. We call these event-delimited, status–status relationships the *interstitial*[2] behavior of the interface. Interstitial behavior is modeled naturally by specification languages that provide for both

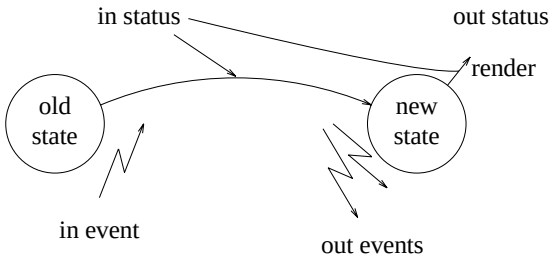

Figure 1: The context for status and event

Notation	Input		Output	
	Event	Status	Event	Status
grammars [22, 23]	√			
STN [17]	√			
process algebras [2, 5, 24]	√		√	
actors [4]	√		√	
CNUCE interactors [21]	√		√	
York interactors [12]	√		√	√
statecharts [14]	√		√	
PIE model [11, 10]	√			√
modified PIE [10]	√	√		√
Z/VDM		√		√
object-oriented Z		√	√	√
* NEW MODEL *	√	√	√	√

Table 1: A classification of user interface specification notationss with respect to status and event.

[1] We distinguish between state and status. State refers to the internal information of an entity, whereas status refers to information externally available. An entity might provide status as output that reflects some of its internal state

[2] We have chosen this term based on a rather archaic, but entirely relevant, use of the term interstice from the *Shorter Oxford English Dictionary*, defined as "An intervening space of time; an interval between actions"

status input and output and event input.

The last entry in Table 1 refers to the new model of specification that we will present in the next section. This classification of previous approaches points quite clearly to the absence of any one approach that treats both status and event information symmetrically.

Before presenting such a model it is worth reiterating why it is necessary to introduce the more complicated notation. Proponents of both event and status based approaches can justifiably argue that they can encompass all types of behavior. For example, it is possible to model a status value in a process algebra, such as CSP:

$$\begin{aligned} StatusVar(v) &= (set?x \rightarrow StatusVar(x)) \\ &\quad \Box \\ &\quad (get!v \rightarrow StatusVar(v)) \end{aligned}$$

It is also possible to model an event, such as a keystroke, as simply the change in a status. Polling devices behave exactly like that. The important observation is that, although both methods 'work' and may even be the way the phenomena is implemented, they do not naturally represent the phenomena. This leads to a specification that is constantly trying to go against the grain of the notation and thus almost certainly to one that is incomplete or wrong.

Take, for example, the behavior of a mouse dragging an icon. When the mouse button is depressed and over an icon, subsequent movement of the mouse also moves the icon until the mouse button is released. We will first look at a CSP description of the mouse alone that distinguishes between dragging and non-dragging behavior.

$$\begin{aligned} NoDrag &= mouseDown \rightarrow Drag \\ &\quad \Box \\ &\quad move \rightarrow NoDrag \\ Drag &= mouseUp \rightarrow NoDrag \\ &\quad \Box \\ &\quad move \rightarrow Drag \\ Mouse &= NoDrag \end{aligned}$$

This description just exhibits the toggling back and forth between the dragging and non-dragging mode of the mouse. Movement of the mouse is allowed in either mode. We did not need to mention explicitly the position of the mouse, nor did we have to be explicit about what constituted a move in terms of an (x, y) displacement. This last abstraction is good because we don't want to make an arbitrary decision about what constitutes a noticeable move at this level of description.

Now we want to coordinate the dragging mode of the mouse with the dragging of an icon. The icon is dragged only when the position of the mouse is within the boundary of the icon. We cannot express this condition without an explicit representation of position for both the mouse and the icon. There are two different conditions under which a *mouseDown* event occurs — one in which the mouse is over the icon and one in which it is not. Dragging only occurs in the first context. In CSP, we would like to be able to express such a condition, but without explicit state, we must use events *on* and *off*.

$$\begin{aligned} NoDrag &= mouseDown \rightarrow on \rightarrow Drag \\ &\quad \Box \\ &\quad mouseDown \rightarrow off \rightarrow NoDrag \\ &\quad \Box \\ &\quad move \rightarrow NoDrag \end{aligned}$$

Depressing the mouse while not on an icon might invoke another mode of the mouse, but we have ignored that additionaly complexity in this example. This description is OK, but it relies on a correct interpretation of the intent of *on* and *off* and the understanding that this event is not actually determined by the mouse or the icon.

We must also indicate that the movement of the mouse is accompanied by an equivalent movement of the icon. In the previous specifications, the *move* events referred only to movement of the mouse. We must introduce another event for the movement of the icon. Furthermore, we must guarantee that the movements are equivalent and that will require that we introduce parameters to the movement events.

$$\begin{aligned} NoDrag &= mouseDown \rightarrow on \rightarrow Drag \\ &\quad \Box \\ &\quad mouseDown \rightarrow off \rightarrow NoDrag \\ &\quad \Box \\ &\quad (\Box_{x,y}\, move(x, y) \rightarrow NoDrag) \\ Drag &= mouseUp \rightarrow NoDrag \\ &\quad \Box \\ &\quad (\Box_{x,y}\, move(x, y) \rightarrow Imove(x, y) \\ &\qquad \rightarrow Drag) \\ Mouse &= NoDrag \end{aligned}$$

We are forced to represent the continuous relationship between mouse and icon position by the sequential ordering of separate movements.

There are still some problems with this solution. For example, if we were to describe the relationship between three objects — a mouse and two icons — so that dragging one icon over another caused the second icon to become highlighted, we would have to introduce even more events that needed correct interpretation to indicate when positions overlapped. Another problem with this description of dragging is that by separating the movements of the mouse and the icon we have introduced the possibility in this notation that other events could intervene and decouple the two movements. That decoupling is certainly not what we intend for dragging, and it would not be discovered until a more complete model of the system were analyzed. We discuss this emergence of unintended behavior later on.

We could try out many variations on the event description of this dragging phenomenon, none of which is all that difficult to write down. But none of the options is immediately obvious as the right choice. This simple example has shown how an event-oriented notation results in specifications that are in the best case cumbersome and in the worst case incorrect. We would have had similar problems if we chose a notation that relied only on status phenomena, for there it would have been difficult to indicate when significant events, such as a mouse click, occur. Status-only descriptions would have to use status as flags which are judiciously set and unset to signal the occurrence of an event. This kind of awkward specification can be seen frequently in model-oriented notations to signal pre-conditions for an operation.

In the next section, we introduce a hybrid status/event language. Part of the example in that section describes the dragging phenomenon as a means of comparison.

3 The hybrid model for status/event description

The derivation of the new specification notation is fairly simple. We will take parts of existing specification techniques and combine them. Description of interstitial behavior will be inherited from the modified PIE model. In addition, we will need to add events as outputs, similar to the interactor models or process algebras.

3.1 Concrete notation

In order to deal with examples we will need a concrete notation for describing these interactive agents[3]. However, we hold this notation lightly as we believe that it is the concepts underlying it that are important, not the particular notation. It will serve to illustrate the utility of our approach.

For each agent, we describe its internal state as a binding from identifier names to types (similar to a model-oriented specification approach). In addition, we list the input and output events in which the entity can engage. Input and output status are described as type bindings, similar to the internal state. The following is a description of a mouse.

MOUSE — signature

state:	none
in-events:	none
in-status:	none
out-events:	*mouse_up*
	mouse_down
	mouse_click
out-status:	$mouse\_x, mouse\_y : \mathbf{N}$

3.2 Specification of a slider

The mouse example is a bit too trivial for demonstration purposes, as it contains no internal state and, hence, no state transitions of interest. The behavior of the mouse is completely described by the continuous changes in its status and event output. A more complex interactive agent involving internal state is a slider control used in most graphical user interfaces. The appearance of the slider is shown in Figure 2.

For this example, we will consider two ways of interacting with the slider.

Dragging While the mouse is held down over the slider background, the slider's handle moves up and down with the mouse. If the mouse is released within the slider background, the handle moves to the relevant position. If the mouse is released outside the scroll area, the slider snaps back to its original position.

Jumping If the mouse is clicked over the scroll area, the handle moves to that position.

We will consider the components of the slider specification by first indicating the static description, or signature, followed by rules defining the behavior associated with events and finally the interstitial behavior between events.

[3] We use the name agent to reflect the extension to Abowd's agent model and notation [1, 2].

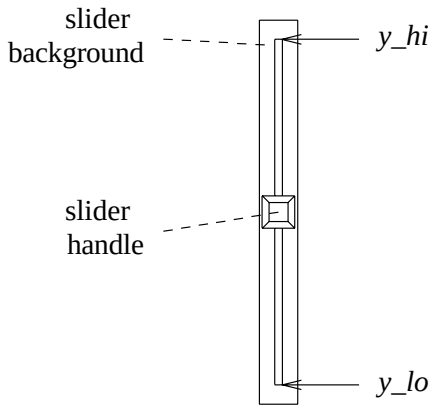

Figure 2: Slider control

3.3 Slider signature

First, we look at the internal state, inputs and outputs of the slider. All of these inputs come from the mouse agent. The state variable *position* indicates the location of the slider handle when it is not being dragged. It will also indicate the position of the handle prior to dragging in case the dragging is aborted with no change. The *background* is a region of point, denoted by $\mathbf{P}(\mathbf{N} \times \mathbf{N})$, that indicates an active area during dragging. The *dragging* variable records when the user is engaged in dragging. This *cannot* be inferred from the state of the mouse buttons and the position of the mouse. One output status value is the same as the internal state *position* value and the other indicates the position of an outline handle during dragging. These output status values can be linked to application agents to determine offsets within a buffer, for example.

SLIDER — signature

state:	$position : 0 \,..\, 1$
	$background : \mathbf{P}(\mathbf{N} \times \mathbf{N})$
	$dragging : Boolean$
in-events:	$mouse\_up, mouse\_down, mouse\_click$
in-status:	$mouse\_x, mouse\_y : \mathbf{N}$
out-events:	none
out-status:	$position, dragging\_pos : 0 \,..\, 1$

We don't explicitly mention the mapping from status values to screen display. The slider handle (shown in Figure 2) is always displayed and is centered at *position*. An outline of the slider handle is displayed at *dragging_pos* only when $dragging = true$. An alternative to the slider we have specified above would have the actual handle moving during dragging, not an outline of it. In this case, the slider would be at *dragging_pos* while being dragged and at *position* otherwise. These are just two options for the status–status mapping between the abstract slider status and the display.

3.4 Slider state transitions

We next look at the behavior of the slider when events occur. A *mouse_down* event initiates dragging, a *mouse_up* event

terminates dragging, and a *mouse_click* event performs a jump scroll. The state transition associated with an event is specified by indicating how internal state values change. Dashed names (indicated with a $'$) represent internal state values after the transition, and undashed names refer to values before the transition. For clarity, we have indicated some predicates below (e.g., "on") in English only.

SLIDER – state transitions

on *mouse_down*:
  if $(mouse\_x, mouse\_y)$ on background
  then $dragging' = true$
  $position' = position$

on *mouse_up*:
  $dragging' = false$
  if *dragging* and
  $(mouse\_x, mouse\_y)$ on *background*
  then $position' = calc\_pos(mouse\_y)$
  else $position' = position$

on *mouse_click*:
  $dragging' = false$
  if $(mouse\_x, mouse\_y)$ on *background*
  then $position' = calc\_pos(mouse\_y)$
  else $position' = position$

The function *calc_pos* translates from screen position (from *y_lo* to *y_hi*) to a slider position (from 0 to 1).

$$calc\_pos : (y\_lo \mathinner{..} y\_hi) \rightarrow (0 \mathinner{..} 1)$$
$$calc\_pos(y) = \frac{(y - y\_lo)}{(y\_hi - y\_lo)}$$

3.5 Interstitial behavior

With the inclusion of status information in this model, we did not have to describe mouse movement in terms of events. Contrast this with the description of movement in Section 2. Instead we can describe the effect of mouse movement more naturally using interstitial constraints. This aspect of the slider is very important as it used to generate the constant feedback that is necessary for the usability of the control.

SLIDER – interstitial behavior

if *dragging* and $(mouse\_x, mouse\_y)$ on *background*
then $dragging\_pos = calc\_pos(mouse\_y)$
else $dragging\_pos = position$

Summary on slider example

The slider example was chosen carefully because even though it is a fairly simple and ubiquitous interaction object, we needed to have access to both status and event information for a simple description. Any specification of such a slider in a language that did not include status and event information would have been more cumbersome or incomplete. The state transitions depended on what input event had occurred, the current state and the status input. The value of the internal state variable *dragging* depends in a somewhat complicated way on the history of user actions. The status output was a function of both the state and the status input. Furthermore, without the status output and the associated interstitial behavior, we would not have been able to capture the user feedback continuously (recall the decoupling problem in Section 2).

It is also instructive to note that explicit definition of the interstitial behavior forces us to be clear about its dependence on both status input (mouse position) and the current state (*dragging*). We cannot simplify the formula to:

SLIDER – interstitial behavior

if $(mouse\_x, mouse\_y)$ on *background*
then $dragging\_pos = calc\_pos(mouse\_y)$
else $dragging\_pos = position$

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*.

4 Addressing mixed control issues

The previous example is a little unrealistic because it assumed that changes to the slider were only initiated by the user. In text editing applications, the position of the slider indicates a relative position of the window viewing the text and we would expect changes in the text to automatically affect the slider position.

4.1 A modification to the slider

In order to accommodate this sort of interaction between the slider agent and some other application agent, we must modify the slider specification of Section 3. There are two ways we could modify the slider. The first method would introduce new input events to update the slider position that would be generated by the application attached to the slider. The other method — which we will examine more closely here — would be to provide the position information as input status that is supplied by the application agent. In this approach, we also have to provide a way for the slider to announce that it has been moved by the user so that it's new position is reflected by a changed offset in the application. The revised signature of the slider is given below.

SHARED_SLIDER — signature

state:	$background : \mathbf{P}(\mathbf{N} \times \mathbf{N})$
	$dragging : Boolean$
in-events:	$mouse\_up, mouse\_down, mouse\_click$
in-status:	$mouse\_x, mouse\_y : \mathbf{N}$
	$position : 0 \mathinner{..} 1$
out-events:	$changed(v : 0 \mathinner{..} 1)$
out-status:	$position, dragging\_pos : 0 \mathinner{..} 1$

In the previous slider specification, all assignments to *position* should be removed, as this information is now provided by a source outside of the slider agent. The $changed(v)$ events must be generated at the appropriate points, specifically after a *mouse_up* or *mouse_click* event. The state transitions are given below.

SHARED_SLIDER – state transitions

on *mouse_down*:
  if (*mouse_x*, *mouse_y*) on slider
    dragging' = *true*

on *mouse_up*:
  dragging' = *false*
  if *dragging* and (*mouse_x*, *mouse_y*) on *background*
    raise *changed*(*calc_pos*(*mouse_y*)

on *mouse_click*:
  dragging' = *false*
  if (*mouse_x*, *mouse_y*) on *background*
    raise *changed*(*calc_pos*(*mouse_y*))

The interstitial behavior does not change for this example.

The changes to the slider specification reflect a different conceptual model of the slider's operation. Since the original slider model could not address the situation in which an external agent other than the user affected the position of the slider handle, we were forced to think of other possible solutions and analyze them. For example, now that we allow the slider position to be adjusted by a user and another application agent, we can ask what would happen if both tried to adjust the position at the same time. In the modified shared slider above, the behavior is well-defined. As the user is dragging the outline of the slider handle, the input status *position* is allowed to change, meaning that the actual handle can move while the user is dragging the outline handle. When dragging is complete, the changed event will be raised and interpreted by the application.

If the slider had been described entirely in terms of events (as would happen in a typical window manager), then the above problem might not have surfaced. There would have been an ad hoc solution that would have *emerged* from the particular event orders, but there would have been no need to explicitly *design* this behavior. A formalism ought to force interface developers to face these critical design decisions.

4.2 Extending to multi-user systems

Ad hoc or emergent solutions to problems such as above might suffice for single-user systems because they represent unlikely occurrences. Users might not have any expectation for the behavior in unusual situations, so we do not have to worry so much about the system conflicting with user predictions. But the example of simultaneous input to a single agent is quite prevalent in a multi-user system. For example, it would be common in a shared editor for one user to be editing the text while another is scrolling using the slider. This situation forces the issue of where control of the slider resides and has serious ramifications on usability. The modified slider specification that models the slider position as status information, provides a sensible solution to the shared activity.

Figure 3 presents a plausible architectural view of the shared editor. In this diagram, event connections between agents are indicated by jagged arrows and status connections are indicated by straight arrows. Figure 4 presents a simple scenario of shared use of a slider in the multi-user context. In that figure, we see the viewport of a single user, Alison, consisting of the text contained inside the round-edged box, the scrollbar and the mouse pointer. The rest of the text

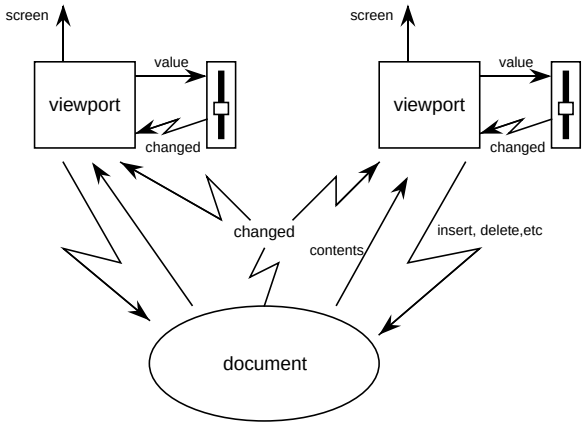

Figure 3: Architecture of shared text editor

is part of the document. Above Alison's viewport, we can see the insertion point for another user, Brian. We see three snapshots of Alison interacting with the slider while Brian types in text. Initially (a), Alison selects the slider handle and drags its outline upwards. While she is dragging the slider (b), Brian types text in the part of the document that is above Alison's view, causing the position of the slider handle to move down. Finally (c), when Alison releases the mouse, her slider announces a *changed* event that results in the slider position and viewport being adjusted. In this scenario, Alison and Brian's interactions do not interfere with each, though they do affect the information displayed in each other's view.

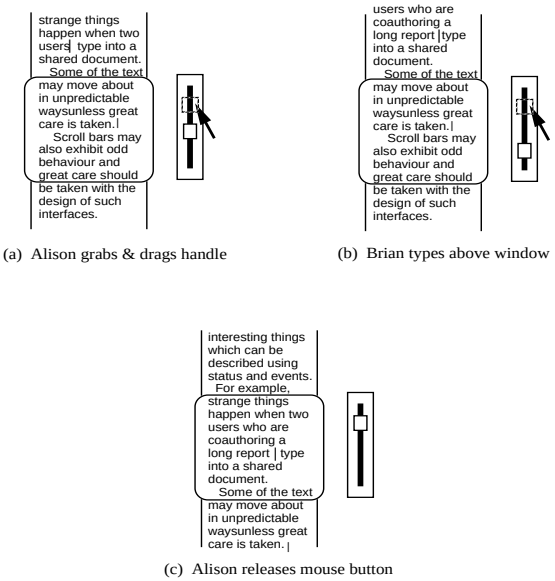

Figure 4: Scenario of shared use of a slider

This slider example in a multi-user context is an instance of the more general problem of shared values with mixed

control. The appropriate use of status–status mappings simplifies the specification of the system and in so doing allows a designer to focus on the necessarily complex issues of shared update. It is not the mere use of a status/event language that provided a clear solution to the problems of mixed control. Rather, the appropriate use of status/event descriptions has allowed us to focus on the central issues and has exposed the nature of the problem. Contrast this again with a purely event-based description. In this case, the behavior would depend on the order in which various events arrived. Even if the designer never considered the problem of mixed control a solution would emerge in the actual use of the system. 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. The behavior of the resulting system is engineered and not simply emergent.

5 Related work and other issues

The status/event language presented in this paper is very much motivated by an approach to specification that borrows the good parts of existing specification languages. The justification for this is that while most specification languages provide enough power to describe all possible behaviors, they all have a bias that makes the description of some behaviors easier than others. Developing specification languages that are a mix of existing languages is not novel. For example, several researchers have recognized the advantage of mixing event-based process algebras with model-oriented notations [1, 5, 18, 20, 24]. Our work is similar in spirit, except that we have concentrated on the phenomena of status rather than internal state, as status is a concept of great importance in describing interactive systems naturally from the perspective of the user.

More recently, Zave and Jackson have coined the term multiparadigm specification for this activity of specification through a mixture of languages with different semantic domains [25]. In that work, the concern is the development of a common semantic domain in which to embed all others. Our emphasis here is on establishing the importance of status and event in interactive system development. The syntax and semantics we present is only of importance as far as it can enlighten the difficult problems of interface design, such as mixed control of shared objects.

It is always an important question to ask how concepts in formal specification relate to any possible implementations. An important concept used in this paper is the status–status mapping that can be used to express interstitial behavior. There are at least two popular implementation mechanisms for realizing these mappings. The first mechanism is the constraint, as used in the Garnet [19], Rendezvous [16] and other systems. Constraints are used for high-level descriptions of user interfaces and they are very similar to status–status mappings. Constraint managers typically use events to mediate the resolution of conflicts, that is, when some user or application behavior breaks a constraint an event signals that the constraints must be reestablished. It is important to understand how the mediation proceeds to know exactly how the constraint will be resolved. But constraint managers do not allow the programmer to see or control the mediation and resolution process. This lack of control is precisely how subsequent problems emerge unintentionally.

For example, suppose 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 or constraints 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 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, 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 $x \rightarrow y \rightarrow z$ or $y \rightarrow x \rightarrow z$, the different variables are temporarily inconsistent with one another, but at any moment each is consistent with the current or previous value of a, and the value of z remains positive. However, if the system were to update in the order $x \rightarrow z \rightarrow y \rightarrow z$, then the intermediate value of z would be -5 — a negative value. If some other part of the system relied on z remaining positive, this intermediate value could cause trouble.

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 order, the third constraint will be repaired next, updating z. Only then would the second constraint be addressed updating y that would again invalidate the last constraint and so finally z would be updated a second time.

A more complex system might be able to detect such sequences, but if the user interface updates were delayed until a consistent state were reached then the latency could become excessive. The point of this example is to demonstrate that status–status mappings require event mediation. In languages that do not have events, such as constraint systems, this mediation is hidden from the designer and can result in undesirable delays or unanticipated inconsistencies.

A second implementation mechanism, used for example in the Suite system [6, 7], is the active variable. Active variables automatically generate appropriate change events in order to sustain status–status mappings. But Suite implements a particular policy for dealing with the issue of mixed control and the designer must decide whether these policies are appropriate or alter various parameters to tune the policies for a given circumstance.

In the programming language arena, the status/event distinction could be compared to the distinction between the imperative and declarative programming paradigms. The combination of these two paradigms has been called constraint imperative programming [13]. Though our motivation for arriving at the status/event distinction is very different, these constraint imperative programming languages

might provide a suitable implementation language for our specification notation.

As we saw in the case of the shared slider, event based implementations of status–status mappings depend on broadcasting change events to other interested agents so that they can repair the mappings. We have recently been experimenting with an implementation paradigm that directly supports this type of event (triggers) allowing interested agents to register callbacks 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.

6 Conclusions

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 deals with both status and event uniformly for input and output. We have shown that it is possible to define a model that 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 in this paper and would be happy to see other styles of notation extended in a similar fashion. The two crucial features that would be in any such notation are definitions of both event induced state transitions and interstitial behavior. Indeed, it is the interstitial behavior, the fluid change between actions, that is largely responsible for the sense of responsiveness in the interface. The use of status/event descriptions exposes several issues that need to be addressed in many interfaces, especially in the design of groupware, as we demonstrated with the examination of mixed control interaction with a scrollbar.

7 Acknowledgments

We would like to thank the anonymous referees for their detailed and constructive remarks that were useful for improvements to an earlier version of this work.

References

[1] Abowd, G. D. Agents: Communicating interactive processes. In *Human-Computer Interaction—INTERACT'90* (1990), D. Diaper, D. Gilmore, G. Cockton, and B. Shackel, Eds., Elsevier Science Publishers, pp. 143–148.

[2] Abowd, G. D. Formal aspects of human-computer interaction. Technical Monograph PRG-97, Oxford University, Programming Research Group, 1991. D.Phil. thesis.

[3] Abowd, G. D., Bowen, J., Dix, A., Harrison, M., and Took, R. User interface languages: A survey of existing methods. Technical Report PRG-TR-5-89, Oxford University Computing Laboratory Programming Research Group, October 1989. Also published as internal report 2487-TN-PRG-1008 Issue 1.0 for ESPRIT project 2487 (REDO).

[4] Agha, G. A. *Actors: a Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[5] Alexander, H. Executable specifications as an aid to dialogue design. In *Human-Computer Interaction — INTERACT'87* (1987), H. J. Bullinger and B. Shackel, Eds., North Holland, pp. 739–744.

[6] Dewan, P. A tour of the Suite user interface software. In *UIST'90: Proceedings of the ACM Symposium on User Interface Software and Technology* (1990), ACM, pp. 57–65.

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

[8] Dix, A., and Abowd, G. Integrating status and event in formal models for interactive systems. In *Formal Methods in Human–Computer Interaction*, M. Harrison and C. Johnson, Eds. Cambridge University Press. Draft chapter in preparation.

[9] Dix, A., Finlay, J., Abowd, G., and Beale, R. *Human–Computer Interaction.* Prentice Hall International, 1993.

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

[11] Dix, A. J., and Runciman, C. Abstract models of interactive systems. In *People and Computers: Designing the interface* (1985), P. Johnson and S. Cook, Eds., Cambridge University Press, pp. 13–22.

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

[13] Freeman-Benson, B.-N., and Borning, A. The design and implementation of Kaleidoscope'90 — a constraint imperative language. In *Proceedings of the 1992 International Conference on Computer Languages* (1992), IEEE Computer Society Press, pp. 174–180.

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

[15] Harrison, M. D., and Duke, D. J. A review of formalisms for describing interactive behaviour. In *ICSE-16 Workshop on Software Engineering and Human-Computer Interactino: Joint Research Issues* (1994), R. N. Taylor and J. Coutaz, Eds.

[16] Hill, R. The Rendezvous constraint management system. In *UIST'93: Proceedings of the ACM Symposium on User Interface Software and Technology* (1993), ACM, pp. 225–234.

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

[18] Marshall, L. S. *A Formal Description Method for User Interfaces.* PhD thesis, University of Manchester, United Kingdom, 1986. Also published as technical report UMCS-87-1-2.

[19] Myers, B. A., Giuse, D. A., Dannenberg, R. B., Zanden, B. V., Kosbie, D. S., Pervin, E., Mickish, A., and Marchal, P. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer 23*, 11 (November 1990), 71–85.

[20] Nielsen, M., Havelund, K., Wagner, K. R., and George, C. The RAISE language, method and tools. *Formal Aspects of Computing 1*, 1 (1989), 85–114.

[21] Paternò, F., and Faconti, G. On the use of LOTOS to describe graphical interaction. In *People and Computers VII, HCI'92 Conference* (1992), A. Monk, D. Diaper, and M. Harrison, Eds., Cambridge University Press, pp. 155–174.

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

[23] Schiele, F., and Green, T. HCI formalisms and cognitive psychology: the case of task-action grammars. In *Formal methods in Human-Computer Interaction*, M. D. Harrison and H. W. Thimbleby, Eds., Cambridge Series on Human-Computer Interaction. Cambridge University Press, 1990, ch. 2.

[24] Sufrin, B., and He, J. Specification, refinement and analysis of interactive processes. In *Formal methods in Human-Computer Interaction*, M. D. Harrison and H. W. Thimbleby, Eds., Cambridge Series on Human-Computer Interaction. Cambridge University Press, 1990, ch. 6.

[25] Zave, P., and Jackson, M. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology 2*, 4 (1993), 379–411.